НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ. ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
(повна назва інституту/факультету)

Кафедра обчислювальної техніки
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“___”_____ 2020 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки 123 Комп’ютерна інженерія
(код і назва)

на тему: Гра у жанрі Платформер

Виконав: студент __4__ курсу, групи __ІО-62___
(шифр групи)

Ставцев Дарій Ігорович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник професор, д.т.н., Жабін В.І. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю професор, д.т.н., Сімоненко В.П. _ _____
(посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент професор, д.т.н., Дичка І. А. _____ _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО”

Факультет Інформатики та обчислювальної техніки

Кафедра Обчислювальної техніки

Освітньо-кваліфікаційний рівень **бакалавр**

Напрям підготовки **123 « Комп’ютерна інженерія »**

ЗАТВЕРДЖУЮ

Завідувач кафедри

Сергій СТІРЕНКО _____
(підпис)

«___»_____ 2020 р.

Завдання
на бакалаврський дипломний проект студента

Ставцева Дарія Ігоровича
(прізвище, ім’я, по батькові)

1. Тема проекту (роботи) Гра у жанрі Платформер

керівник проекту (роботи) Жабін В. І., д.т.н., професор,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «__» _____ 2020 року №1180-С

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вихідні дані до проекту (роботи) *технічна документація*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)
Розробка комп’ютерної гри у жанрі Платформер за допомогою ігрового рушія Unity, використовуючи мову програмування C# та середу розробки Visual Studio для роботи з даною мовою. Опис та класифікація предметної області, аналіз існуючих розробок, дослідження ігрових рушіїв, визначення алгоритму та функціоналу проекту, реалізація прототипу.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень):

Структурна схема алгоритму, діаграма класів, Use-case діаграма.

6. Консультанти проекту (роботи), з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	д.т.н, проф. Сімоненко В.П.		

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів дипломного проекту (роботи)	Строк виконання етапів проекту(роботи)	Примітки
1.	*Затвердження теми роботи*	*01.09.2019-01.09.2019*	
2.	*Вивчення та аналіз завдання*	*15.12.2019-15.03.2020*	
3.	*Написання вступної частини*	*15.03.2020-25.03.2020*	
4.	*Огляд існуючих прототипів*	*25.03.2020-5.04.2020*	
5.	*Програмна реалізація*	*5.04.2020-15.04.2020*	
6.	*Оформлення документації дипломного проекту*	*15.04.2020-26.05.2020*	
7.	*Передзахист*		
8.	*Захист*		

Студент-дипломник Дарій СТАВЦЕВ

Керівник роботи Валерій ЖАБІН

АНОТАЦІЯ

В бакалаврській дипломній роботі метою було створення гри у жанрі Платформер. Для досягнення мети було проаналізовано жанрову класифікацію ігор, проаналізовані сучасні представники цього жанру, а також були досліджені різні середи розробки ігор та ігрові рушії. На основі цього було реалізовано прототип двовимірного платформера. У ході розробки були написані скрипти, необхідні для роботи гри, та створенні або запозиченні графічні та звукові матеріали.

Для розробки відповідної програми використано ігровий рушій Unity 2019, що використовує мову C# та засобом програмування алгоритму є середа розробки Visual Studio 2019.

ANNOTATION

In the bachelor's thesis, the goal was to create a game in the genre of Platformer. To achieve this goal, the genre classification of games was analyzed, modern representatives of this genre were analyzed, as well as various game development environments and game engines were studied. Based on this, a prototype of a two-dimensional platformer was implemented. During the development, the scripts needed to run the game and create or borrow graphic and audio materials were written.

To develop the corresponding program, the game engine Unity 2019 is used, which uses the C# language and the tool for programming the algorithm is the development environment Visual Studio 2019.

Технічне завдання

ЗМІСТ

1. Найменування та область застосування 2
2. Підстави для розробки 2
3. Мета та призначення розробки 2
4. Джерела розробки 2
5. Технічні вимоги 2
6. Етапи та стадії розробки 3

1. Найменування та область застосування

Дане технічне завдання стосується створення комп'ютерної гри у жанрі Платформер. Прототип гри, що розробляється, може бути використаний для загального користування або у статтях та інших проектах з посиланням на даний дипломний проект.

2. **Підстави для розробки**

Підставою для розробки є завдання на виконання бакалаврської дипломної роботи, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут» імені Ігоря Сікорського.

3. **Мета та призначення розробки**

Метою розробки є прототип гри у жанрі Платформер.

4. **Джерела розробки**

Джерелами розробки є науково-технічна література, технічна документація, публікації та статті у мережі Інтернет.

5. **Технічні вимоги**

Оскільки для розробки проекту був обраний ігровий рушій Unity, а для написання і редагування програмного коду використовується Microsoft Visual Studio 2019, то знадобиться обладнання, що володіє технічними характеристиками не нижче мінімально необхідних для коректної роботи даних засобів розробки. Тому для реалізації проекту був використаний ноутбук DELL Inspirion 5559 з наступними технічними характеристиками:

- Операційна система Windows 10;
- Процесор Intel Core i5-U6200, 2.4Ghz;
- Об'єм оперативної пам'яті 4 GB;

- Відеокарта AMD Radeon M335 4 GB VRAM.

6. **Етапи і стадії розробки**

Назва етапів виконання	Термін виконання
Затвердження теми роботи	01.09.2019-01.09.2019
Вивчення та аналіз завдання	15.12.2019-15.03.2020
Написання вступної частини	15.03.2020-25.03.2020
Огляд існуючих прототипів	25.03.2020-5.04.2020
Програмна реалізація	5.04.2020-15.04.2020
Оформлення документації диплом-ної роботи	15.04.2020-26.05.2020
Передзахист	
Захист	

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЕКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1.	А4		Завдання на дипломний проект	2	
2.	А4	ІАЛЦ.622500.002 ТЗ	Технічне завдання	3	
3.	А4	ІАЛЦ.622500.003 ПЗ	Пояснювальна записка	65	
4.	А3	ІАЛЦ.622500.006 А1	Структурна схема алгоритму	1	
5.	А3	ІАЛЦ.622500.008 А2	Діаграма класів	1	
6.	А3	ІАЛЦ.622500.009 А3	Use-case діаграма	1	
7.	А4	ІАЛЦ.622500.010 Б1	Лістинг програми	30	

Пояснювальна записка до дипломного проекту

ЗМІСТ

ВСТУП 4

РОЗДІЛ 1 5

ДОСЛІДЖЕННЯ КЛАСИФІКАЦІЙ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР 5

1.1 Аналіз та загальна характеристика комп'ютерних ігор 5

1.2 Аналіз існуючих розробок 9

1.2.1 Dead Cells 10

1.2.2 Cuphead 12

1.2.3 Mark of the Ninja 13

1.2.4. Starbound 14

Висновок до розділу 1. 16

РОЗДІЛ 2 17

РОЗРОБКА ПРОЕКТУ 17

2.1 Аналіз середовищ розробки та обґрунтування вибору технології розробки проекту 17

2.1.1 Unreal Engine 4 17

2.1.2 Cry Engine 5 18

2.1.3 Unity 2019 23

2.2 Загальний алгоритм реалізації проекту 26
2.3 Аналіз потенційної аудиторії споживачів 30
2.4 Актуальність проекту 30
2.5 Мета проекту 31
2.6 Функціонал проекту 31
Висновок до розділу 2 33
РОЗДІЛ 3 34
РЕАЛІЗАЦІЯ ПРОЕКТУ 34
3.1 Графічне оформлення 34
3.1.1 Спрайти ігрового героя 35
3.1.2 Спрайти інтерфейсу 35
3.1.3 Спрайт ворогів 37
3.2 Проектування гри 37
3.2.1 Фізичні властивості об’єктів 38
3.2.2 Рух об’єктів 38
3.2.3 Анімації для героя 40
3.2.4 Зброя героя 41
3.2.5 Гибель та відродження героя 43
3.2.6 Штучний інтелект ворогів 44
3.2.7 Графічний інтерфейс для героя та ворога 45
3.2.8 Взаємодія між об’єктами 47
3.2.9 Генерація хвиль ворогів 48
3.2.10 Графічний інтерфейс до генератору хвиль 51
3.2.11 Умови для завершення гри 53

3.2.12 Меню початку та кінця гри 53
Висновок до розділу 3. 56
РОЗДІЛ 4 58
ІНСТРУКЦІЯ КОРИСТУВАЧА 58
4.1 Стартове меню 58
4.2 Інтерфейс та ігровий процес 58
4.3 Умови та меню завершення гри 61
Висновок до розділу 4. 62
ВИСНОВКИ 63
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ: 64

ВСТУП

Індустрія комп'ютерних ігор з'явилась зовсім нещодавно, але вже займає одну з лідируючих позицій на світовому ринку. Це має просте пояснення: стрімкий розвиток технологій починаючи з кінця 20-го століття та поява Інтернету. Завдяки цьому, в порівнянні з іншими видами розваг, комп'ютерні ігри стали більш доступними та популярними. На сьогоднішній день розробка нових комплектуючих для комп'ютера тісно пов'язана з індустрією ігр, бо якість та вимоги ростуть, що сприяє розвитку комп'ютерних технологій. Також варто прийняти до уваги, що комп'ютерні ігри це не лише один з видів розваг та відпочинку сучасної людини. Наприклад, багато сучасних інтернет ресурсів використовують комп'ютерні ігри для навчання, створюються комплекси для симуляції, які використовуються для навчання спеціалістів різних напрямків та професій.

В країнах СНД, нажаль, розвиток ігрової індустрії не досягає світового рівня. Перші розробки та загальна культура комп'ютерних розваг прийшла до нас пізно та не мала належних спеціалістів та навичок для її розвитку. Тому тема розробки ігор є дуже актуальною для країн СНД та потребує розвитку та появи нових спеціалістів та компаній-розробників, щоб мати можливість конкурувати з зарубіжними розробниками та виходити на світовий ринок.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ КЛАСИФІКАЦІЙ ТА АНАЛІЗ ІСНУЮЧИХ РОЗРОБОК КОМП'ЮТЕРНИХ ІГОР

1.1 Аналіз та загальна характеристика комп'ютерних ігор

Комп'ютерна гра – комп'ютерна програма, що служить для організації ігрового процесу зв'язку з партнерами по грі, або сама виступає в якості партнера. Під час комп'ютерної гри за допомогою спеціальних програм створюється імітація прямої взаємодії у віртуальному просторі між ігровим персонажем та користувачем (або групою користувачів) за певним алгоритмом. Спостерігаючи ігрову ситуацію на екрані монітора, гравець впливає на неї за допомогою клавіатури, «миші», джойстика, тощо.

Комп'ютерні ігри часто створюються на основі сторонніх джерел, таких як книги та фільми, але останнім часом існують приклади і зворотного напрямку, коли на основі відомої гри чи ігрової серії починають з'являтися додаткові матеріли, що розширюють всесвіт гри.

Більш того, спеціально розроблені ігри можуть виступати в якості навчального матеріалу або дозволяють використовувати гравців в науково-дослідних цілях. За деякими популярними іграми проводяться змагання різних видів масштабності, від регіональних до світових, які мають окрему назву «кіберспорт».

Комп'ютерні ігри мають настільки істотний вплив на сучасне суспільство, що останнім часом з'являється стійка тенденція до гейміфікації для неігрового прикладного програмного забезпечення.

Таким чином, в деяких європейських навчальних закладах почали використовувати відомі ігри для навчання, а для потреб армій створюються симулятори для тренуванния солдатів. Деякі країни зараховують кіберспорт до офіційного виду спорту, та, наприклад, уряд США у 2011 році визнав комп'ютерні ігри окремим видом мистецтва, поряд із театром та кіно.

З усього цього можна зробити висновок, що комп'ютерні ігри щільно влились до нашого нинішнього життя. Сфера їх використання за останні роки продовжує зростати, ігри використовують не тільки для розваг, але й для проведення наукових досліджень та навчання.

Внаслідок того, що критерії приналежності гри до того чи іншого жанру не визначені однозначно, класифікація комп'ютерних ігор недостатньо систематизована, і в різних джерелах дані про жанр конкретного проекту можуть розрізнятися. Проте, існує консенсус, до якого прийшли розробники ігор, і приналежність гри до одного з основних жанрів майже завжди можна визначити однозначно.

За основний критерій поділу жанрів беремо дії, які найбільш часто здійснюються в іграх цього жанру. Ігри діляться на три великі групи: ігри контролю, ігри дії та ігри інформації. Всі ігри, що входять до групи дуже схожі між собою, але одночасно з цим мають різні відхилення.

Виділено 15 основних геймплейних елементів з яких складається взагалі будь-яка гра: навчання, загадки, спілкування, роль, вивчення, збирання, ухилення, знищення, змагання, техніка, турбота, розвиток, контроль, тактика, планування. Ці жанри були поділені на 3 різні класифікації. (Табл. 1.)

Таблиця 1 - Жанрова класифікація комп'ютерних ігор.

Категорія	Ігри інформації	Ігри дій	Ігри контролю
Гібридні жанри	• Action-RPG • Roguelike	• MMOFPS • Survival	• RTS • MOBA
5 елементів	• Open RPG	• Open Action	• Global Strategy
3 елемента	• RPG • MUD MMORPG	• Action • Slasher • Battle Racing	• Strategy • Sim Strategy

Продовження *Таблиці 1* - Жанрова класифікація комп'ютерних ігор.

2 елемента	• Puzzle • Quest • Browse RPG • Adventure	• Platformer • Stealth-Action • Fighting • Racing	• Economical • Tower Defense • Wargame • Cardgame
1 елемент	• Education • Test • Contact • Hero • Toure	• Arcade • Horror • Shooter • Sport • Simulator	• Logic • Tactic • MicroControl • Building • Life Sim
Елементарні жанри	1.1. Навчання 1.2. Загадки 1.3. Спілкування 1.4. Роль 1.5. Вивчення	2.1. Збирання 2.2. Ухилення 2.3. Нищення 2.4. Змагання 2.5. Водіння	3.1. Турбота 3.2. Створення 3.3. Контроль 3.4. Тактика 3.5. Планування

Ігри інформації: Як зрозуміло з назви, головне в іграх цієї групи - отримання інформації в усіх її проявах. В таких іграх іноді присутній і планування, і динаміка, але інформація в них найважливіше. Золотою серединою групи є «RPG (RolePlaying Game)» - «рольова гра».

Ігри, в яких можна жити, вживатися в роль героя, в яких в якості головних достоїнств виставляють атмосферу, сюжет, ігровий світ.

Ігри дії: Головне в іграх цієї групи - руху, які необхідний здійснювати керуючи якимось тілом (людським або гуманоїдним), або технічним засобом. Золотою серединою групи є «Action» (гра-бойовик). Найбільш динамічні

ігри. Саме ці ігри прийнято хвалити за те, що вони розвивають швидкість реакції.

Ігри контролю: Група «гри контролю» складається з тих ігор, головна суть яких - планування подій і управління для досягнення переваги в подальшому. Сюди потрапляють всі види стратегій, різні економічні ігри, варгейми, тактики. Золотою серединою групи є «Strategy» (звичайна локальна стратегія).

Усі ігри поділяються на одиночні та мультиплеєрні. Також мультиплеєрні ігри поділяються між собою на:

- Мультиплеєр на одному комп’ютері;
- Мультиплеєрні оффлайн-ігри;
- Масові онлайнові.

Саме завдяки цій класифікації встановлюються режими, що будуть присутні у грі. Одиночна гра – вид гри, у яких приймає участь одна людина. Зазвичай гравцю протистоїть штучний інтелект, а його метою є рух до кінця гри (проходження), накопичення ресурсів або прокачування навичок. Часто ці цілі комбінуються.

Іншим видом ігр є мультиплеєр. Цей режим гри пристосований до гри більше ніж однієї людини одночасно. В багатьох іграх одиночна гра там мультиплеєр комбінуються.

За візуальною складовою комп’ютерні ігри поділяються на:

- Текстові – гра з мінімальною кількістю графічних елементів, а спілкування з гравцем відбувається за допомогою тексту;
- 2D – усі елементи гри розроблені за допомогою двовимірної графіки;
- 3D – усі елементи гри розроблені за допомогою тривимірної графіки.

Після ретельного аналізу класифікацій комп’ютерних ігр, було прийнято рішення розробити двовимірну гру у жанрі Платформер. Ігри цього жанру

мають зрозумілу та просту механіку, що є безпосередньо плюсом для починаючих розробників. Також двовимірна гра не потребує великої кількості людей та ресурсів для розробки, на відміну від ігор тривимірної графіки.

1.2 Аналіз існуючих розробок

Платформер — жанр ігор, ігровий процес якого складається з стрибків героя по різноманітним платформам та через перешкоди. Також примітним елементом цього жанру є колекціонування предметів, що необхідні для проходження рівня. В платформерах гравець керує персонажем, який рухається платформами різних розмірів, висоти та масштабів в залежності від типу гри та її складності. Традиційно у перших двовимірних платформерах персонаж рухався лише зліва направо, але вже в сучасних іграх головний герой може рухатися у будь-якому напрямку.

Деякі предмети, що гравець збирає протягом гри, наділяють керованого гравцем персонажа додатковими бонусами, які в свою чергу можуть буди як і вичерпними (на певний рівень чи на короткий час), так і до повного проходження гри. Предмети, зброя та різні ігрові бонуси зазвичай збираються звичайним дотиком персонажа і для застосування не вимагають виконання спеціальних дій з боку гравця. Предмети збираються в «інвентар» персонажа та застосовуються за допомогою спеціальних команд. Вороги у платформерах представлені у вигляді так званих чисельних та різноманітних «монстрів» із примітивним штучним інтелектом, завдяки якому намагаються максимально наблизитися до гравця та нанести йому шкоду, або не володіють штучним інтелектом зовсім, переміщаючись по певній заданій траєкторії або здійснюючи повторювані дії. У багатьох випадках зіткнення з ворогом призводить до втрати очок життя або смерті персонажа. Іноді вороги задані тікати, видавати звуки або змінювати свій зовнішній вигляд при наближенні до гравця. У перших платформерах вороги нейтралізувались звичайним стрибком на них, але на сьогодні майже в усіх платформерах головний герой володіє якоюсь

зброєю для боротьби з монстрами. Переможені монстрами зазвичай зникають або провалюються за ігрові елементи.

Рівні часто мають приховані проходи в стінах та заховані у важкодоступних місцях предмети, які можуть істотно полегшити гравцю проходження гри.

Майже усі ігри цього жанру характеризуються фентезійними елементами та мальованою графікою. Головними героями таких ігор можуть бути як і звичайні люди у незвичайних обставинах, так і міфічні істоти (наприклад гноми, дракони) чи антропоморфні тварини.

Першу активність жанр почав проявляти на початку 1980-х років, з виходом на той час дуже популярних платформерів Pitfall та Super Mario Bros, поява яких сприяла розвитку усього ігрового жанру. Поява Pitfall принесла до індустрії зміну в системі проходження рівнів з вертикальної до горизонтальної, що залишилось і до нашого часу. Натомість Super Mario Bros є прикладом для творців ігор, завдяки розмірам та складності ігрових рівнів.

Оскільки технології з кінця 1980-х отримали стрімкий розвиток, аналізувати будемо саме сучасні приклади платформерів, такі як:

- Dead Cells;
- Cuphead;
- Mark of the Ninja;
- Starbound.

1.2.1 Dead Cells

Комп’ютерна гра у змішаному жанрі roguelike та платформер, розроблена французькою студією Motion Twin для операційних систем Windows, MacOS, Linux, та для консолей Nintendo Switch, PlayStation 4 і Xbox One пізніше. (Рис 1.1.)

У Dead Cells гравець контролює істоту, яка намагається вибратися із лабіринту. Рівні у грі генеруються процедурно, по ним розкидані вороги та різні скарби, у тому числі і зброя з випадковими характеристиками. Подібно до ігор у жанрі roguelike, персонаж Dead Cells має лише одне життя – якщо він помре, то гравець буде повинен почати гру спочатку. Деякі навички, отримані в одному проходженні переносяться і на наступні проходження.

У ході гри персонаж досліджує різні підземелля, перемагає ворогів, що їх населяють, та збирає багаточисельні предмети. Персонаж може носити з собою декілька видів зброї, та знаходити в підземеллях нові – від мечей і метальних ножів до гранат, капканів та турельних установ. Зброя та предмети мають випадкові характеристики та змінюються від складності гри.

Кожен рівень у кожному підземеллі генеруються випадковим чином. Гра сама збирає складні лабіринти с заготовлених раніше елементів, та випадково розкидає по ним предмети та ворогів. Гра передбачає, що персонаж буде часто помирати, а гравець вчитись на помилках.

Dead Cells отримала дуже високі оцінки критики – приємна візуальна частина, складний ігровий процес та висока якість гри у цілому.

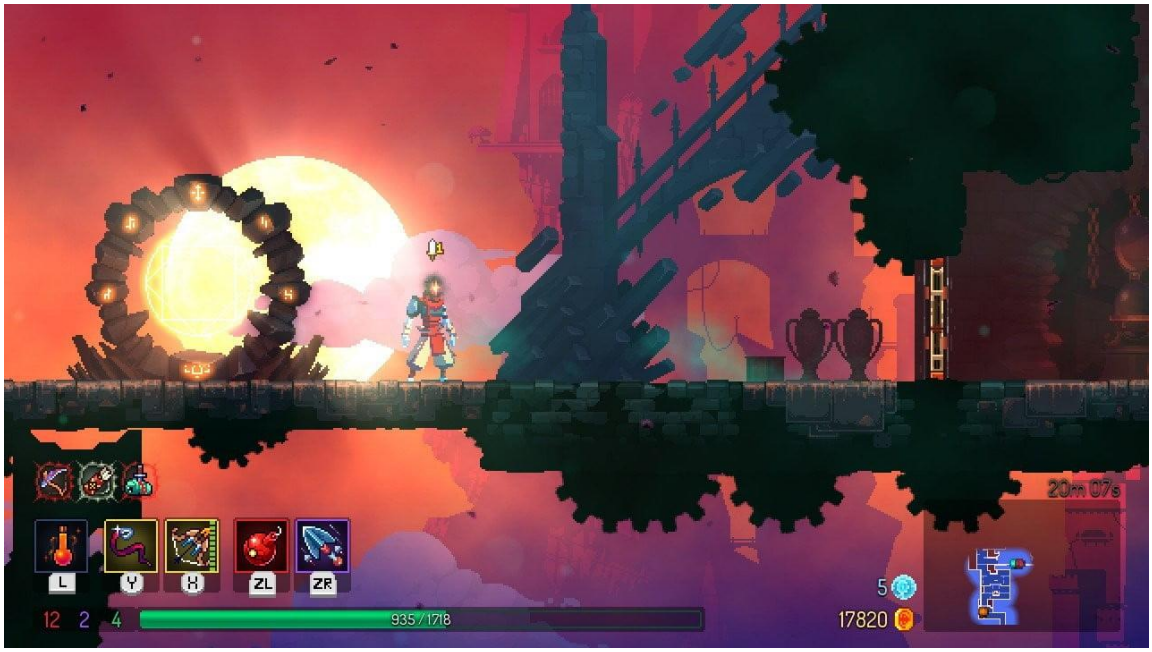


Рис 1.1 Знімок екрана з гри Dead Cells.

1.2.2 Cuphead

Відеогра у жанрі платформера та shoot ‘em up, розроблена та видана канадською студією StudioMDHR Entertainment у 2017 році для персональних комп’ютерів та Xbox One на базі ігрового рушія Unity.

Гра була натхненна роботами аніматорів Disney 1930-х років та виконана у схожому стилі, намагаюсь зберегти характерні ознаки сюрреалістичної анімації.

Головний герой, Cuphead – це хлопчик з чашкою замість голови, що за допомогою пострілів із пальца бореться з ворогами. Програвши суперечку з дияволом, Cuphead повинен пройти через велику кількість босів, щоб повернути борг дияволу та отримати назад свою душу.

Доступ до рівнів у грі відбувається за допомогою мапи зовнішнього світу, що виконана у стилі ігор жанру Action-RPG (Рис 1.8), сама мапа є розгалуженою послідовністю рівнів, а також має свої таємні проходи та магазин, у якому персонаж може придбати додаткові життя та різні допоміжні навички для боротьби з ворогами.

Крім звичайних пострілів з пальца, герой Cuphead також має навички парування різного виду атак та об’єктів за допомогою долоні, а успішне парування накопичує шкалу, що дозволяє використовувати інші спеціальні вміння. У цілому бойова система в грі складається у багаточисельних битв с босами у стилі ігор жанра shoot’em up («біжи та стріляй») , де кожен бос має свої унікальні навички, а після перемоги на екран виводиться статистика з успіхами гравця, що дозволяє при бажані перемогти боса ще раз для підвищення своїх результатів.

Гра Cuphead також має кооперативний режим гри, в якому приймає участь ще один гравець управляючи персонажем Mugman, та допомагаючи головному герою у битвах з ворогами. (Рис 1.2)

Рис 1.2 Знімок екрану з гри Cuphead.

1.2.3 Mark of the Ninja

Комп’ютерна гра у жанрі платформер з елементами стелс-екшну, розроблена канадською студією Klei Entertainment для персональних комп’ютерів та Xbox One у 2012 році. (рис 1.3.)

Сюжет гри розповідає про ніндзю в якого немає ім’я, але він носить титул Чемпіона клану Хісомі. Головний герой має спеціальне тату, яке посилює сприйняття, рефлекси та реакцію, але поступово зводить його з розуму. Він намагається звільнити членів клану, що вижили після нападу ворожого клану Гессен.

Основний ігровий процес заснований на прихованому проходженні рівня, уникненні зустрічі з ворогами та знешкодженні пасток. Головний герой також може ховатись за різними елементами інтерфейсу. Зброя чемпіона – меч-ніндзято, який він використовує для безшумного ліквідування жертви, та бамбукові дротики, які використовуються для відволікання та знешкодження джерел світла. Незважаючи на наявність смертельної зброї, гра в рівній мірі орієнтована як на усунення всієї охорони рівня, так і на безкровне прохо-

дження. В процесі проходження купується і відкривається додаткове спорядження атакуючого та відволікаючого типу. Практично за кожну дію нараховуються бали, на які можливо відкрити нові прийоми, придбати нове або поліпшити старе спорядження.

Дана гра є відмінним стелс-екшеном серед двовимірних ігор подібного жанру, яких не так багато проводиться останнім часом. Володіє стильним графічним оформленням, яке задає атмосферу усій історії.

Рис 1.3 Знімок екрана з гри Mark of the Ninja.

1.2.4. Starbound

Комп’ютерна гра платформер із змішаним жанром пісочниці, розроблена та видана студією Chucklefish Games у 2016 році для портативних комп’ютерів, PlayStation та Xbox One, написана цілком на мові програмуванні C++ та має власний ігровий рушій. (Рис 1.4)

За сюжетом гравець потрапляє у процедурно згенерований світ з великою кількістю планет, на поверхні яких можна висаджуватись, будувати будівлі, добувати різні ресурси та шукати скарби. На початку гри гравцю пропонуються на вибір 7 рас, які кардинально відрізняються одне від одного: зовнішнім виглядом, рівнем технологічного прогресу, культурою, режимом

правління, стилем життя та методами колонізації. Starbound починається з появи головного героя на своєму космічному кораблі, а в залежності від обраної змінюється й мотив героя, чому він покинув рідну планету. Гра містить багато завдань та сюжетних ліній, які приводять гравця на різні планети, а космічний корабель головного героя виступає у ролі транспорту для переміщення по всесвіту.

Кожна планета та її елементами, такі як: тип поверхні, погодні умови, тип матеріалів, навички ворогів, гравітація, колір будівель та ресурсів, та інша велика кількість ігрових елементів генеруються випадково.

Рис 1.4 Знімок екрана з гри Starbound.

Висновок до розділу 1.

Після аналізу існуючих розробок, можливо зробити висновок, що ігор, які включають себе лише один жанр Платформера дуже невелика кількість, через те що інші ігрові жанри також розвивались і платформери перестали відповідати вимогам гравців. Однак не можна говорити, що ігри цього жанру втратили свою популярність. На сьогоднішній день виходить маса різних платформерів, які можна зарахувати і до інших жанрів. Це може говорити про те, що «чисті платформери» стали продуктом направленим на вузький круг споживачів, тому для отримання великих прибутків розробники почали додавати до цього жанру елементи сторонніх ігрових жанрів.

Тому можливо зробити заключний висновок, що для успіху гри її потрібно урізноманітнити елементами різних жанрових категорій.

РОЗДІЛ 2
РОЗРОБКА ПРОЕКТУ

2.1 Аналіз середовищ розробки та обґрунтування вибору технології розробки проекту

Ігровий рушій – програмне забезпечення, призначене для розробки комп’ютерних ігор та інших інтерактивних програм, що обробляють графіку у режимі реального часу.

Для аналізу було взято на розгляд декілька ігрових рушіїв – Unreal Engine 4, CryEngine 5, Unity 2019.

2.1.1 Unreal Engine 4

Ігровий рушій, що розробляється та підтримується компанію Epic Games. Першою грою на цьому рушії став шутер від першої особи Unreal у 1998 році. (Рис 2.1.). Спочатку був призначений для розробки шутерів від першої особи, але його наступні версії успішно застосовувалися в іграх самих різних жанрів, в тому числі стелс-іграх, файтингах та і масових багатокористувацьких рольових онлайн-іграх.

Рис 2.1 Знімок екрана з гри Unreal.

Unreal Engine підтримує більшість існуючих платформ та операційних систем – Windows, Linux, MacOS, iOS, Android, Xbox One, Xbox 360, PlayStation 2, PlayStation 3, PlayStation 4, GameCube, Wii, Wii U, Nintendo Switch, PSP, Dreamcast.

Також для комфортної роботи з редактором Unreal Engine персональний комп'ютер має задовольняти мінімальні системні вимоги:

- Версії операційних систем повинні бути не нижче ніж Windows 7/8/10 64x, Linux Ubuntu 15.04 та MacOS 10.13;
- Процесор Intel Quad-core чи AMD (з частотою 2.5Hz та вище);
- Оперативну пам'ять 8 GB (для операційних систем Linux 16 GB);
- Відеокарта AMD Radeon 6870 HD або NVIDIA GeForce 470 GTX та вище.

Рушій має зручний та простий інтерфейс, що вдосконалюється з кожною новою версією. (Рис 2.2.)

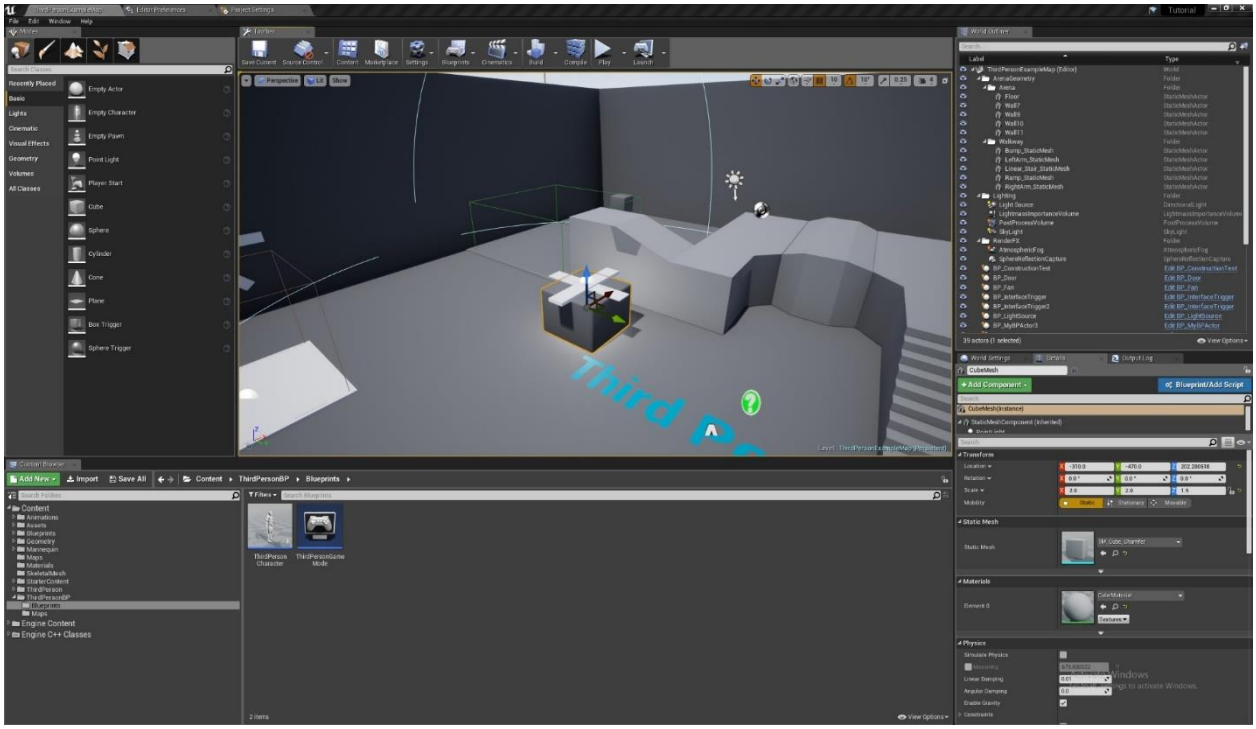


Рис 2.2 Інтерфейс Unreal Engine 4.

За допомогою великої кількості налаштувань тіней, освітлення та рендеру, рушій дозволяє домогтися фотореалістичної графіки. Також інструментарій рушія істотно спрощує роботу з елементами ландшафту та рослинності.

Ігрову логіку можливо розроблювати й без написання коду за допомогою системи візуальних скриптів Blueprint, але більшість розробників використовує мову програмування C++.

За допомогою вбудованого редактору візуальних ефектів Cascade, з можливістю налаштування системи частинок з використанням різних модулів. Також рушій має редактор матеріалів, який використовує штучне затінення та дає повний контроль над зовнішнім виглядом об'єктів та персонажів. Використовується широкий набір анімаційних інструментів для редагування сітки та анімації. Отриманий результат роботи можливо відразу подивитись та при потребі редагувати.

Для створення відеороликів та різних сцен з анімацією вбудовано Sequencer, інструмент, що дозволяє налаштовувати персонажів, камеру та освітлення.

Unreal Engine також працює з режимом віртуальної реальності, з розширеними настройками управління рухом. Компанія-розробник Epic Games співпрацює із світовими лідерами розробки програмного та апаратного забезпечення, тому Unreal Engine має високу якість взаємодії з системами віртуальної та доповненої реальності. Розширена система штучного інтелекту дозволяє налаштовувати реалістичну поведінку персонажів гри.

Офіційний сайт рушія має велику кількість документації та навчальної інформації для починаючих розробників. Також присутній магазин який має різні матеріали для розробки: моделі персонажів на об'єктів, скрипти, звуки, анімації та різні додаткові плагіни до рушію.

З 2015 року Unreal Engine має вільну ліцензію та може бути використаний ким завгодно, але якщо проект розроблений на базі рушію приносить

прибуток більш ніж 3 тисячі доларів за квартал, розробник має перераховувати 5% від прибутку компанії Epic Games.

Рушій Unreal Engine більш підходить до глобальних розробок ігор у стилі 3D, ніж 2D, але все ж таки має широкий вибір платформ для готових проектів та розробки нових.

2.1.2 Cry Engine 5

Остання версія ігрового рушія розробленого та виданого компанією Crytek. Даний рушій має відмінну графіку. Першою грою, яка була випущена на Cry Engine стала популярна Far Cry. (Рис 2.3.)

Рис 2.3 Знімок екрану з гри Far Cry.

Хоча Cry Engine 5 заявлено як кросплатформерний рушій, він підтримує лише 4 платформи – Windows, Xbox One, PlayStation 4 та Oculus Rift.
Для роботи з ігровим рушієм потрібні відносно невисокі характеристики комп'ютера:

- Персональний комп'ютер з операційною системою Windows 7/8/10 64x;
- Процесор Intel Dual-core (або інший від 2GHz);

- Оперативна пам’ять 4 GB;
- Відеокарта NVIDIA GeForce 450, AMD Radeon HD 5750 (або вище з підтримкою DirectX 11).

Рушій простий та зрозумілий інтерес. (Рис 2.4.)

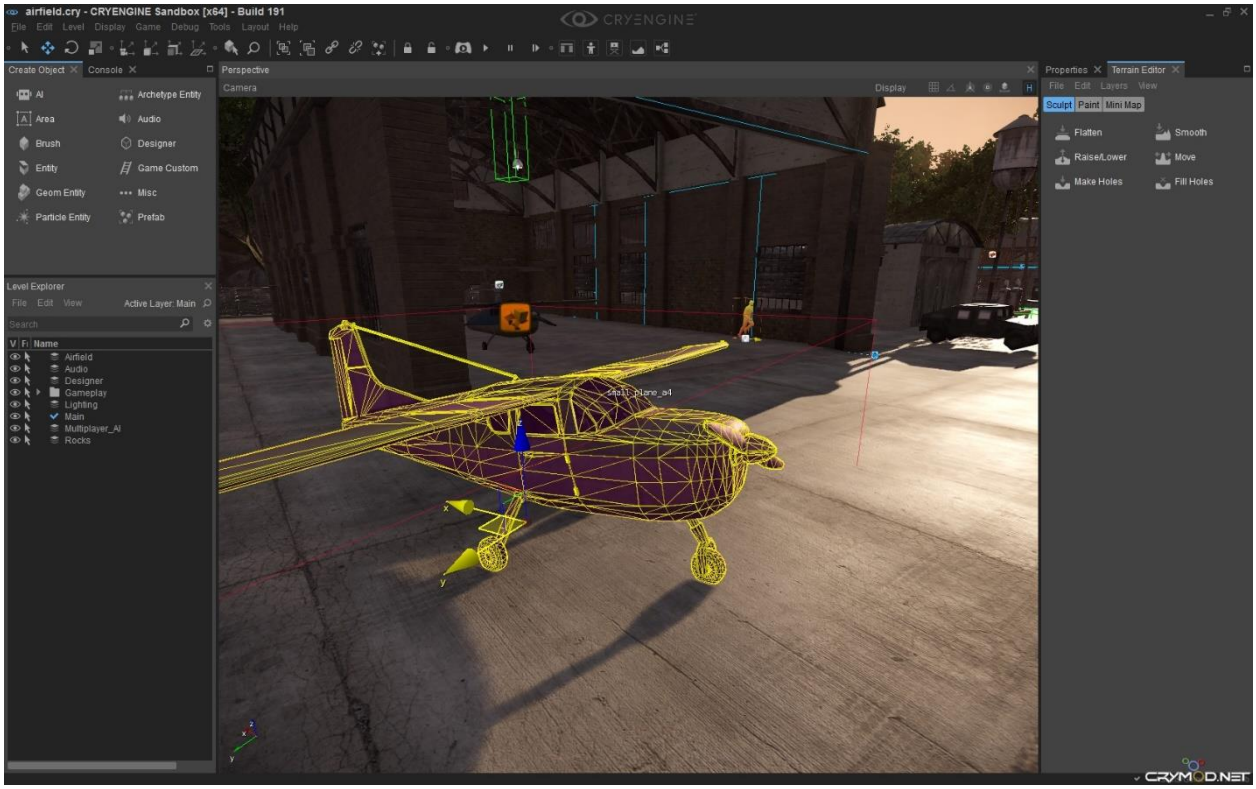


Рис 2.4 Інтерфейс рушія Cry Engine 5.

У Cry Engine зроблено акцент на візуальну складову проекту, тому у ньому присутній різній функціонал на інструментарій для досягнення найкращої якості зображення. Також рушій підтримує останню версію DirectX 12, набору API що відповідає за візуальну складову гри.

Physically Based Rendering – система у Cry Engine, що дозволяє імітувати взаємодію світла з матеріалами та використовує фізику реального світу, що надає правдоподібності вигляду об’єктів. Також за допомогою динамічної оброки водних каустик досягається висока якість зображення води. Даний рушій має інструменти для ефективного згладжування та зниження пікселіза-

ції зображення. Присутня глобальна настройка освітлення, яка дозволяє використовувати більший тональний діапазон, ніж інші рушії, за допомогою якого можна поліпшити розпізнавання світлих, середніх та темних тонів без особливих зусиль.

Для роботи з матеріалами рушій має вбудований редактор Cry Engine SandBox, який пропонує великий інструментарій для створення рівнів та ігрових світів.

За допомогою інструменту Designer Tool є можливість редагувати моделі та експортувати створені матеріали у зовнішні інструменти. Також існує редактор TrackView, що дозволяє створювати відеозаписи та інтерактивні сцені з об’єктами.

Так само як і у Unreal Engine 4, у Cry Engine присутня система візуальних скриптів FlowGraph, яка дозволяє створювати та контролювати ігрову логіку без написання коду та скриптів вручну.

Рушій має велику кількість налаштувань для анімації персонажів, в тому числі параметричну кісткову анімацію. Є вбудована розширена система штучного інтелекту, що дозволяє налаштовувати реалістичну поведінку персонажів.

Cry Engine має можливість аналізувати продуктивність прямо під час гри, а вбудований інструмент Statoscope (Рис 2.5.) дозволяє у графічному вигляди слідкувати за витратою ресурсів комп’ютера для майбутньої оптимізації.

Рис 2.5. Інструмент Statoscope.

Так само як і його попередник, рушій Cry Engine має офіційний сайт з навчальним матеріалом та магазином з різними матеріалами для розробки.

Рушій має вільну ліцензію та може бути використаний любим, але якщо продукт створений на базі Cry Engine приносить компанії чи фізичній особо прибуток понад 5 тисяч доларів у рік, то 5% від цього прибутку повинен бути перерахований компанії Crytek. Так само як і Unreal Engine 4, Cry Engine є відмінним інструментом для розробки глобальних проектів у 3D стилі.

2.1.3 Unity 2019

кросплатформерний рушій для розробки ігор, розроблений компанією Unity Technologies. Unity підтримує майже усі платформи – Windows, Linux, MacOS, Android, iOS, FireOS, PlayStation Vita, PlayStation 4, Xbox One, Nintendo Switch, Nintendo 3DS, Oculus Rift, Steam VR, Gear VR, PlayStation VR, Android TV, Smart TV, TvOS.

Для роботи з Unity потрібен комп’ютер з мінімальними вимогами:

- Персональний комп’ютер з операційною системою Windows 7/8/10 x64 або MacOS X 10.9+;
- Процесор з підтримкою SSE2;
- Відеокарта з підтримкою DirectX 10.

Як і більшість рушіїв, Unity має простий та зрозумілий для розробників інтерфейс. (Рис 2.6.)

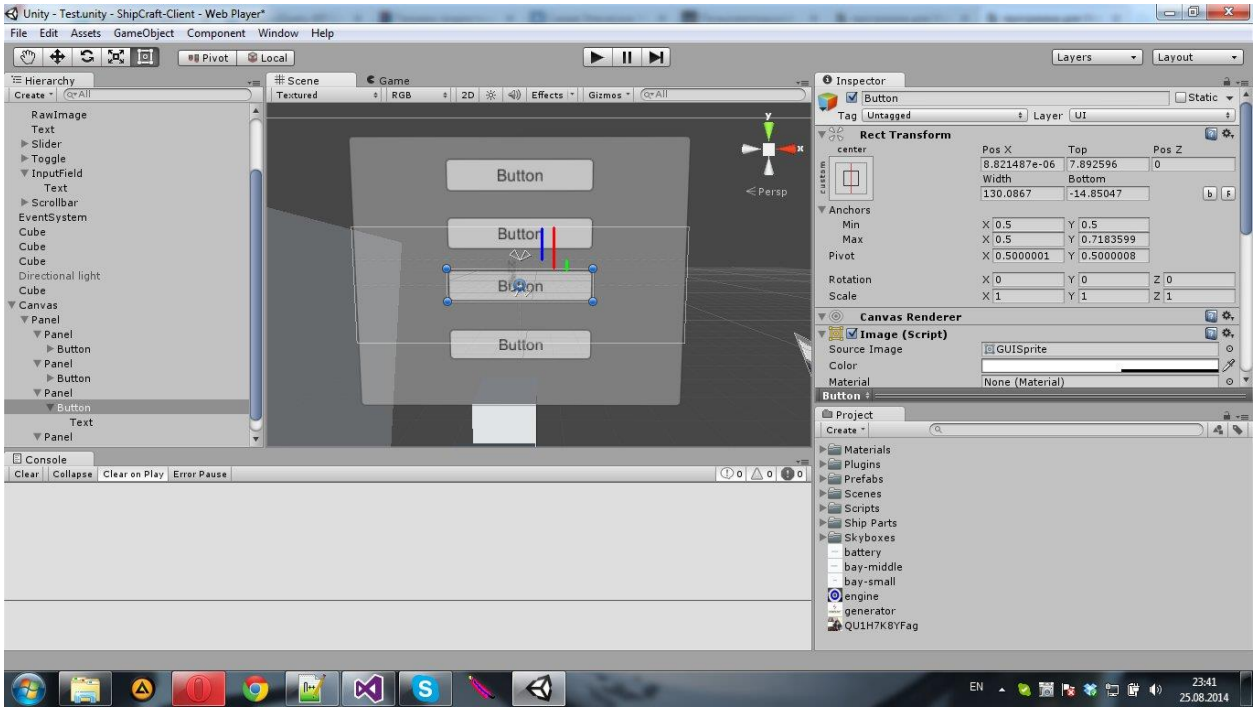


Рис 2.6 Інтерфейс Unity.

Unity є різносторонньою програмою розробки як для програмістів так і для художників завдяки різним інструментам, таким як: Timeline – для розробки анімаційних сцен, Cinemachine – набір динамічних камер, Progressive Lightmapper – для роботи з освітленням, Autodesk Maya – робота з моделюванням та 3D анімацією. Рушій дає можливість розроблювати проекти як у 2D, так і у 3D стилях. Також є підтримка рушіїв NVIDIA PhysX для роботи з фізикою об’єктів та Box2D для роботи з двовимірними об’єктами.

Unity відомий відмінною оптимізацію, що помітно позначається на якості та швидкості праці проекту. Рушій підтримує мови програмування такі як C# та UnityScript.

Як і попередники, що були проаналізовані вище, сайт Unity має дуже велику кількість документації різного направлення, у тому числі і навчаючої. Саме навчаюча документація представлена у вигляді окремих уроків на розборів різних тем.

Unity має свій відомий магазин Asset Store, у якому представлена велика кількість матеріалів для розробки: спрайти, моделі, скрипти, фонові рисунки, різні доповнення та розширення до клієнту Unity, тощо.

Даний рушій має 3 версії придбання: Personal (безкоштовна), Plus (35$/місяць), Pro (125$/місяць). Версії відрізняються лише пропонованим функціоналом та максимально допустимим річним прибутком (100 тисяч доларів, 200 тисяч доларів та без обмежень відповідно). Рушій є дуже різностороннім та унікальним, що дозволяє використовувати його для розробки різноманітних продуктів.

Так як для реалізації нашого проекту головними критеріями вибору середи розробки є вільна ліцензія, навчальна документація, зручний та зрозумілий інтерфейс, низькі системні вимоги та інструменти для розробки 2D проектів, нашим вибором стане ігровий рушій Unity.

Реалізація програмного коду у Unity відбувається за допомогою Microsoft Visual Studio, яка має можливість інтеграції додаткового функціоналу для Unity. (Рис 2.7.)

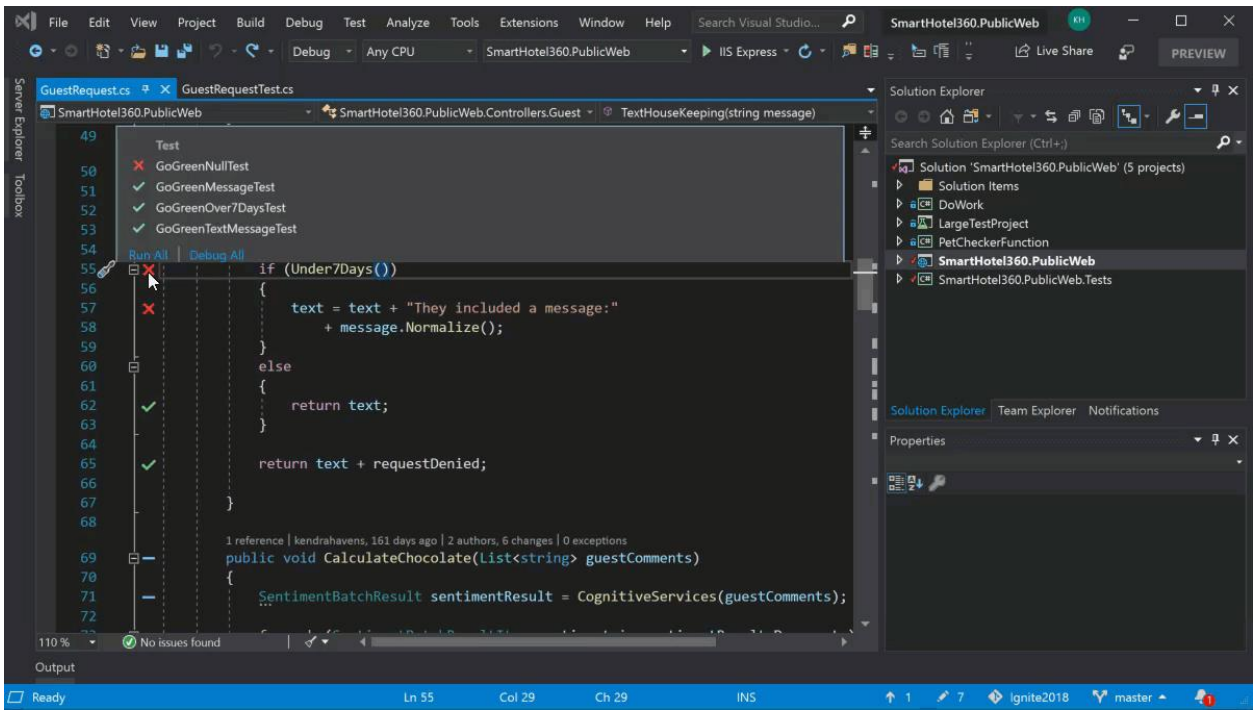


Рис. 2.7 Інтерфейс Visual Studio.

Visual Studio Community 2019 – інтегрована середа розробки програмного забезпечення розроблена та видана компанією Microsoft. Visual Studio включає у себе редактор вихідного коду та має можливість рефакторінгу. Також існує редактор форм для створення графічного інтересу додатків. Дана середа розробки є зручним інструментом програмування та відмінно підходить для написання коду.

2.2 Загальний алгоритм реалізації проекту

Загальний алгоритм розробки комп’ютерної гри мало чим відрізняється від алгоритму розробки будь-якого іншого програмного продукту і включає в себе 3 великих етапи:

- Проектування;
- Розробка;
- Видання та підтримка.

На етапі проектування визначаються мета гри та засоби її розробки.

При визначенні мети виділяються ідея, жанр і сеттинг гри. Ідея - це те, що буде спонукати гравця грати в створювану гру, і вона дуже тісно пов’язана з жанром. Так, наприклад, основна ідея RPG - дозволити гравцеві прожити свою роль так, як він бажає, а основна ідея шутера - дозволити гравцеві взяти участь в реальних чи вигаданих бойових діях. Таким чином, визначивши основні ідеї гри, жанр буде підібраний практично відразу.

Визначившись з жанром і ідеєю гри, наступним кроком буде вибір сеттингу. Сеттинг - це середовище, в якому буде відбуватися основна дія гри. Він визначає місце, час і умови дії. вибір сеттинга може сильно полегшити розробку сценарію для гри, тому його краще вибирати заздалегідь і грунтуючись на смаки цільової аудиторії.

До засобів розробки в першу чергу відносять програмний код та ігровий рушій. Від вибору засобів розробки залежить як швидкість самої розробки, так і працездатність самого продукту надалі. Програмний код в першу чергу залежить від платформи, для якої буде створюватись комп’ютерна гра. Наприклад, якщо гра створюється для браузерів, то логічно буде використання мови Java або Flash, але, якщо гра створюється для персонального комп’ютера, оптимальним вибором буде, наприклад, мова програмування C #.

Ігровий рушій відповідає за опис фізики об’єктів, правил рендеринга графіки, тощо. При виборі ігрового рушія спершу дивляться на його доступність і вже зроблений вибір мови програмування. Наприклад, ігровий рушій Unity дозволяє розробляти гри на мові C # і він є безкоштовним.

Після вибору мети гри і засобів розробки, починається другий етап реалізації проекту - розробка.

Розробка найбільший та найдовший етап реалізації проекту, він включає у себе велику кількість кроків, без яких неможливо створити працездатний продукт.

Насамперед потрібно визначитися з сюжетом та механікою ігрового процесу. Ігрова механіка грунтується на цілі гри, вона визначає всі об’єкти і правила, за якими гравець буде взаємодіяти з ними. Зазвичай паралельно з розробкою ігрової механіки йде написання сюжету гри. Сюжет грає не останню роль, він визначає те, наскільки буде гравцеві цікаво грати у вашу гру. Сюжет представляють в двох варіантах: літературний і режисерський сценарій. літературний сценарій описує основні події і персонажів гри, які беруть участь в грі. Режисерський ж являє собою докладний опис рівнів гри, подій, які на цих рівнях відбуваються.

Так само на даному етапі починається рання опрацювання графічної складової і дизайну гри. На основі сюжету і заздалегідь обумовленого дизайну, створюються ранні концепт-арти, на основі яких згодом буде опрацьовано основний вид гри і персонажів.

Після розробки сюжету і ігрової механіки починається найважливіша частина - розробка самої гри.

На основі сюжету і концепт-артів починається створення персонажів і об’єктів гри, паралельно з цим йде розробка ігрових рівнів. При розробці ігрових рівнів спочатку створюється його спрощений план, на якому схематично зображено сам рівень, а так же зображені предмети, з якими буде згодом взаємодіяти гравець.

Слідом після цього створюється перша версія рівня. Зазвичай, вона являє собою просто голу локацію, з мінімумом необхідних для проходження предметів. Ця версія рівня служить для того, що б протестувати рівень на прохідність. Після тесту, рівень починають поступово заповнювати іншими об’єктами.

Незабаром після створення перших рівнів триває складання першого прототипу гри, який називають альфа-версією гри. Вона необхідна для того,

що б розробник міг провести тестування (альфа-тестування) основної механіки гри, і перевірити наскільки вона відповідає заявленим вимогам. Часто у альфа-версіях гри, у об’єктів навіть немає текстур або вони взагалі представлені у вигляді абстрактних об’єктів.

Якщо альфа-версія гри успішно проходить тестування, настає наступний етап розробки - опрацювання механіки і об’єктів гри.

На даному етапі йде доробка рівнів і механіки гри, і починають додавати перші сюжетні події в гру, такі як відеоролики, сюжетні діалоги і кат-сцени. Так само виправляються перші помилки і несправності в коді гри, які були виявлені при тестуванні альфа-версії гри.

Після цього настає етап створення другого прототипу гри, або, як прийнято говорити, бета-версії. Бета-версія служить для того, що протестувати гру на несправності, фактично бета-версія являє собою практично готову гру. У ній можуть бути відсутні які-небудь незначні елементи, які не впливають на ігровий процес гри. при тестуванні бета-версії гри перевіряється абсолютно все. Часто, особливо, якщо гра є мультиплеєрною, на тестування запрошуються звичайні гравці це дозволяє сильно прискорити час проведення тестування, а так само зняти частину навантаження з команди розробки.

Якщо гра проходить бета-тестування, вона відправляється на остаточне доопрацювання і в правлення критичних помилок, після чого йде збірка фінальної версії гри і слідом настає реліз гри.

Після релізу гри подальша її підтримка. Підтримка полягає у випуску патчів (файлів виправлень помилок в готовому продукті). Так ж для того, що б продовжити життєвий цикл гри, для неї випускають додатковий контент у вигляді DLC, які додають різні предмети і / або можливості для гравця.

Таким чином, стало зрозуміло, що етапи розробки комп’ютерних ігр мало відрізняються, від етапів розробки будь-якого іншого програмного продукту.

2.3 Аналіз потенційної аудиторії споживачів

Характеристика цільової аудиторії є одним із значних етапів розробки будь-якого проекту, так як це допоможе розробнику краще зрозуміти цільову аудиторію і, отже, більш уважно поставитися до тим чи іншим деталям продукту, що розробляється. Тому для проекту була описана його цільова аудиторія.

Для даного проекту, що розробляється була виділена досить велика цільова аудиторія, а саме люди від 15 до 40 років. Вибір такої великої категорії людей обумовлюється відразу декількома причинами. Перш за все, вибір такої категорії людей обумовлений середнім віком активно граючих в комп'ютерні ігри людей, який у країнах СНД стабільно тримається в діапазоні від 30 до 40 років. Саме з цих міркувань була обрана верхня межа цільової аудиторії гри.

Цільовою аудиторією гри були обрані люди, які хочуть познайомитися з даним жанром. Гра не буде володіти високою складністю, але при цьому дозволяє зрозуміти основні механіки, властиві жанру платформер.

Загалом, даний проект може зацікавити найрізноманітнішу аудиторію.

2.4 Актуальність проекту

У наші дні ігрова індустрія розвивається досить швидко, з кожним роком - кількість активних гравців збільшується та розширюється їх аудиторія, з'являються нові студії і нові проекти. Все це приносить величезний прибуток. З часом мали активний розвиток напрямки інді-розробки, що сприяло появі великої кількості незалежних розробників, багато з яких створюють різні цікаві проекти.

Можливо точно сказати, що розроблюваний проект буде мати інтерес серед виділеної цільової аудиторії. Цьому сприяють вибрані жанр і стилізація

проекту. Жанр платформер досить популярний серед гравців, тому що являє собою простий і зрозумілий, а так само цікавий ігровий процес. Володіючи низьким порогом входження, такі ігри доступні широкому колу людей. Так само варто пам’ятати, що сучасне суспільство перенасичене великими проектами, що вимагають від гравця глибокого розуміючи механіки ігрового процесу. Для їх проходження найчастіше потрібна досить велика кількість часу. На їх фоні гра з простим і зрозумілим ігровим процесом має більш зацікавити людей, які не мають можливості приділяти багато часу іграм.

2.5 Мета проекту

Можна виділити наступну мету проекту - аналіз засобів розробки комп’ютерних ігор з подальшою розробкою комп’ютерної гри у жанрі платформер. Створення продукту з використанням основних відмінних рис жанру. Знайомство потенційної цільової аудиторії з не найпоширенішим, але цікавим жанром. Також розробка гри допомагає отримати цінний досвід, який може стати в нагоді не тільки при розробці ігор, але і в багатьох інших областях, а досвід роботи з движком, отриманий під час розробки, дозволяє прискорити роботу при роботі з наступними проектами.

2.6 Функціонал проекту

Проект являє собою комп’ютерну гру жанру двовимірний платформер, основна мета якої полягає в розвазі та наданні засобу для відпочинку та приємного проводження часу.

Перш за все, проект повинен відповідати наступним вимогам:

- Гра повинна володіти простим і зрозумілим ігровим процесом;
- Гра повинна мати просте управління;
- Основна механіка гри полягає в можливості гравця стрибати та бігати по платформам, стріляти у ворогів для уникнення зустрічі з противником або для їх ліквідування;

- Вороги з’являються у певних місцях, але при виявленні гравця вони повинні слідкувати за ним та нападати;
- На рівні повинні бути присутніми об’єкти, що можна зібрати. Ці об’єкти повинні збільшувати рахунок гравця;
- При проходженні гри або при програші гравця, має виводитися невелике меню, яке пропонує або повторити рівень, або перейти до головного меню.

Дані вимоги до функціональної частини є основними, та повинні бути виконані в першу чергу.

Висновок до розділу 2

У даному розділі було детально досліджені середовища розробки ігор такі як Unreal Engine 4, Cry Engine 5, Unity 2019. Був проведений аналіз технічних характеристик для роботи з даними ігровими рушіями, а також було проаналізовано їх інтерфейс, наявність навчальної документації та можливості доступу до роботи з середовищами. Оскільки для розробки нашого прототипу було обрано ігровий рушій Unity, був проведений аналіз середовища Microsoft Visual Studio, яке інтегрується з клієнтом Unity та у якому проводиться уся робота з програмним кодом проекту.

Після дослідження середовищ розробки, визначався основний алгоритм розробки комп'ютерної гри. Були описані ключові моменти створення будь-якого проекту, розглянуто різні етапи створення гри, від написання сценарію до підтримки гри після її випуску. Після цього був проведений аналіз цільової аудиторії для нашого проекту, а також визначено актуальність проекту та його мету. Завершенням розділу 2 є визначення основного функціоналу проекту, яким він має володіти по завершенню розробки.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОЕКТУ

3.1 Графічне оформлення

Основною складовою частиною будь-якої комп'ютерної гри є її візуальна складова. Перед початком опису частини графічного оформлення, слід розібрати основні поняття, що використовуються у геймдизайні, такі як спрайт та тайл.

Тайл (з англ. Tile) – повторюваний фрагмент невеликих розмірів, який служить для створення зображень великих розмірів, а сам цей процес має назву Тайлова графіка. Методи тайлової графіки використовуються для створення рівнів у двовимірних та тривимірних іграх.

Спрайт (з англ. Sprite) – у двовимірних іграх є графічним зображенням якогось об'єкту, та може містити у собі декілька зображень, з яких можна зробити анімацію до об'єкту.

Для реалізації графічної складової нашого прототипу було взято вже готовий набір двовимірних спрайтів з магазину Unity Asset Store, який містить у собі спрайти декількох жанрів та типів ігор на вибір розробника, тому було вибрано спеціальні спрайти та текстури, що підходять до нашого проекту. Для роботи зі спрайтами Unity має вбудовану функцію Sprite Editor (Рис. 3.1.)

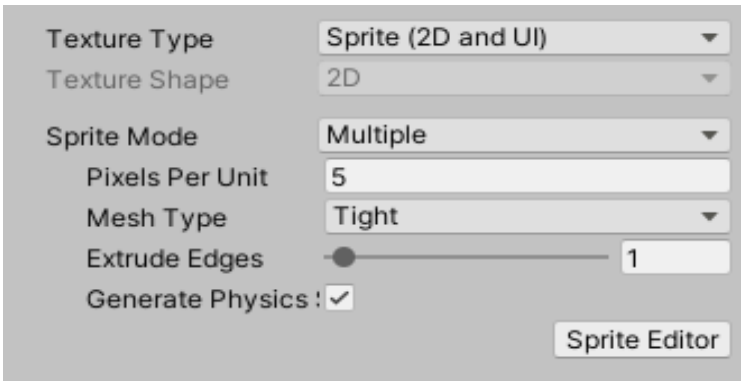


Рис 3.1 Вікно функції Sprite Editor.

Spride Editor дозволяє відкривати великий спрайт-об’єкт, щоб розділити його на складові, для створення анімацій у наступних кроках.

3.1.1 Спрайти ігрового героя

У якості персонажа у нашій гри буде використано спрайти астронавта, який має окремі спрайти для створення анімації простою, бігу, стрибків, а також окремий спрайт для зброї астронавта, яка буде також використана у грі для боротьби головного героя з ворогами. (Рис 3.2.)

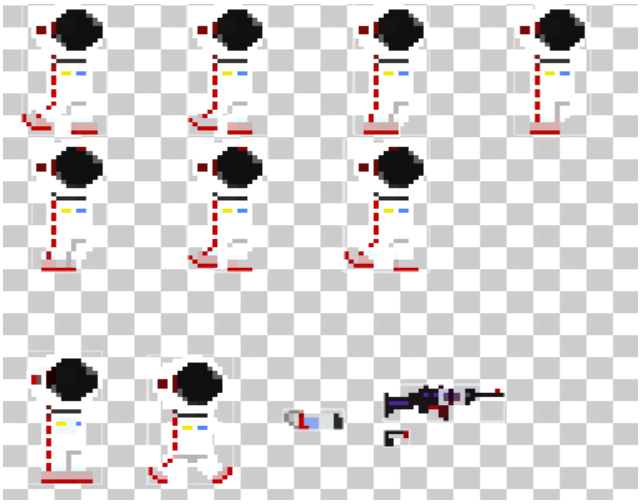


Рис 3.2 Спрайти головного героя.

3.1.2 Спрайти інтерфейсу

Після ігрового персонажа наступним кроком слід підібрати спрайти для фонового зображення, а також елементи візуальної складової гри.

У якості фонового зображення було підібрано нейтральний синій колір, який буде використаний одразу як і для фону сцени на який будуть накладатися інші візуальні елементи, так і для асоціації з небом у грі. (Рис 3.3.)

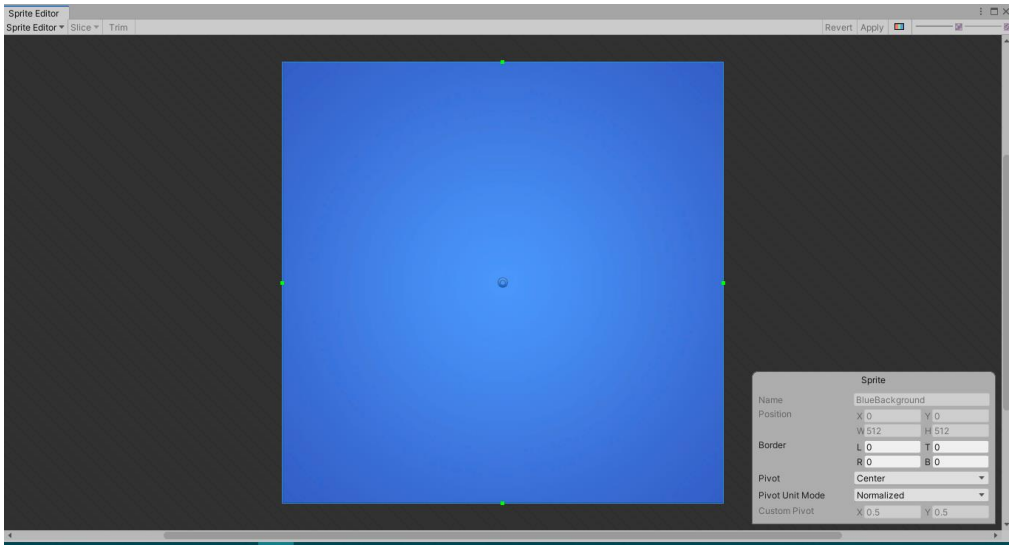


Рис 3.3 Фонове зображення.

Щодо складової візуальних елементів гри, було використано спрайти гір різного кольору, які додають до гри атмосфери прибуття астронавту на якійсь невідомій ворожій планеті. (Рис 3.4.) Самі спрайти також були поділені за допомогою функції Sprite Editor.

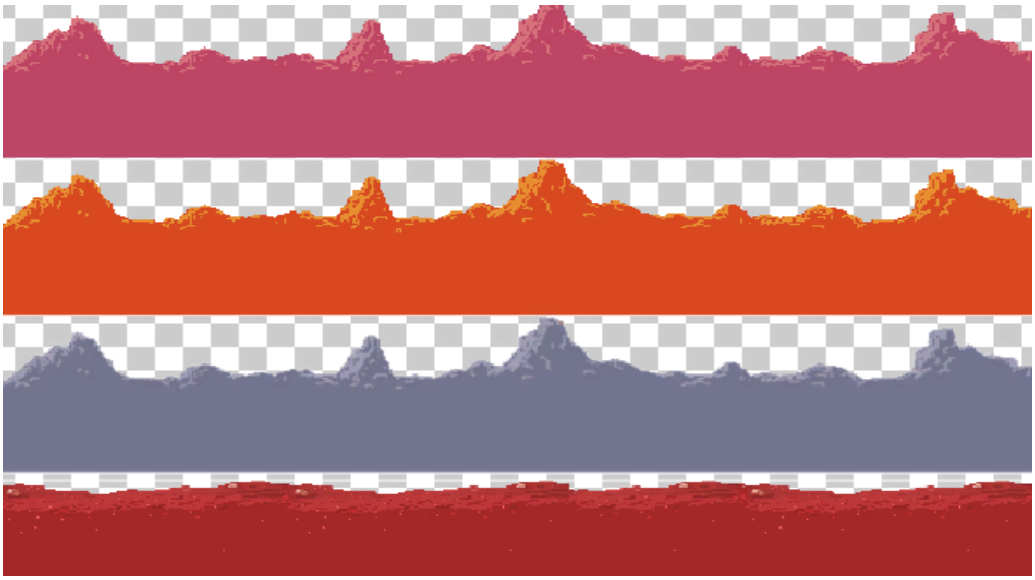


Рис 3.4 Спрайти гір.

Оскільки наша мета створення саме платформера, то для реалізації ігрового процесу потрібно також підібрати спрайти для саме платформ, по яким буде стрибати та бігати ігровий персонаж.

Було підібрано спрайти платформ однієї структури та кольору, які мінімально відрізняються зовнішнім виглядом, але дуже добре підходять до атмосфери створюваного світу (Рис 3.5.)

Рис 3.5 Спрайти платформ.

3.1.3 Спрайт ворогів

Останнім елементом візуальної складової гри є спрайти ворогів, які будуть нападати на ігрового персонажа. Було обрано простий спрайт літаючої тарілки з іноземною істотою у середині. (Рис 3.6.)

Рис 3.6 Спрайт ворога.

3.2 Проектування гри

З візуальною складовою було вирішено, тому можна перейти до поетапного проектування елементів гри.

Перед початком програмування та написанням скриптів, потрібно вирішити проблему з накладанням візуальних елементів гри один на одного, для цього було використано функцію Tags & Layers, яка дозволяє призначити для кажного елементу свій шар, та регулювати який шар буде відображатись поверх іншого для запобігання виникнення візуальних проблем.

3.2.1 Фізичні властивості об’єктів

Першим етапом у створенні прототипу гри є надання ігровим об’єктам фізічних властивостей. Перш за все, нам потрібно надати фізичні властивості платформам, по яким буде рухатись персонаж, адже без них програма не буде рахувати платформи за фізичний об’єкт, а ігровий персонаж буде нескінченно падати. Фізичні властивості об’єктам надаються за допомогою колайдерів, спеціальних компонентів, які надають форму предмету для зіткнення з іншими об’єктами.

Тому, за допомогою функціоналу Unity спершу ми надаємо кожній створеній платформі компонент Box Collider 2D, який надає платформам фізичну форму кубу. Після виявлення фізичних властивостей платформ, потрібно зробити теж саме і для ігрового персонажа, але з деякими нюансами. Оскільки платформи будуть нерухомими, а ігровий герой буде виконувати якісь дії, спершу ми надаємо йому компонент Capsule Collider 2D, який працює так само, як і Box Collider 2D для платформ, але має форму капсули. Головний компонент, який повинен мати ігровий персонаж, це Rigidbody 2D який надає рушію можливість працювати з об’єктом як з фізичною одиницею, а також за допомогою коду надавати різні властивості ігровому персонажу, що є вступом до нашої саме програмної частини проекту.

3.2.2 Рух об’єктів

Наступним кроком розробки ігрового персонажа є написання скрипту для руху та стрибків. Спершу задаємо змінні:

```
    Rigidbody2D rb;
    public float speed;
    public float jumpHeight;
    public Transform groundCheck;
    bool isGrounded;
```

Змінна rb буде приймати значення класу Rigibody2D, який використовуються як основа для виконання фізичних дій об’єктом, змінні speed та jumpHeight використовуються для регулювання розробником швидкості та сили стрибка відповідно. Особлива увага наділяється змінним groundCheck та isGrounded.

Оскільки спершу ігровий об’єкт може стрибати нескінченну кількість разів угору, розробник повинен встановити деякі правила. За допомогою створення пустого об’єкту з ім’ям groundCheck та прикріплення його як дочірнього до об’єкту Player, ми можемо звертатися до об’єкту groundCheck у скрипті.

```
void CheckGround()
    {
        Collider2D[] colliders = Physics2D.OverlapCircleAll(groundCheck.posi-
tion, 0.2f);
        isGrounded = colliders.Length > 1;
    }
```

За допомогою функції CheckGround ми перевіряємо, якщо у певному заданому радіусі від об’єкту groundCheck буде знаходитись інший колайдер, (наприклад, Box Collider 2D, який ми використовуємо для платформ) то виконується перевірка, чи стоїть наш ігровий персонаж на поверхні, яка у свою чергу обмежує стрибки персонажа.

Далі ми повинні зробити так, щоб персонаж рухався. Звертаємось до класу Input, який відповідає за ввід, та його методу GetAxis, який відповідає за рух персонажа по горизонталі. Методу GetAxis(“Horizontal”) приймає значення -1, якщо персонаж рухається вліво, та 1, якщо рухається вправо. Помноживши це на змінну speed, ми регулюємо, з якою швидкістю буде рухатись наш об’єкт.

```
  rb.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, rb.veloc-
ity.y);
```

Але цього недостатньо, бо об’єкт рухається, але дивиться лише у праву сторону, тому ми створюємо наступну функцію:

```
void Flip()
    {
        if (Input.GetAxis("Horizontal") > 0)
            transform.localRotation = Quaternion.Euler(0, 0, 0);
        if (Input.GetAxis("Horizontal") < 0)
            transform.localRotation = Quaternion.Euler(0, 180, 0);
    }
```

Якщо метод GetAxis(“Horizontal”) приймає значення більше 0 (об’єкт рухається вправо), то метод transform.localRotation, який відповідає за поворот об’єкту, надає йому значення по осі y 0, що змушує ігрового персонажа дивитись праворуч. І аналогічно, якщо метод GetAxis(“Horizontal”) буде приймати значення менше 0, методом transform.localRotation по осі y об’єкт прийме значення 180 і персонаж буде дивитись ліворуч.

Після того, як ми змусили нашого ігрового персонажа рухатись, його треба навчити стрибати. Для цього пишемо:

```
if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
            rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
```

Знову звертаючись до класу Input, але вже до методу GetKeyDown, виконується перевірка, якщо натиснена клавіша Space, то за допомогою методу rb.AddForce об’єкт наділяється гравітаційною силою та стрибає, а помноживши це на змінну jumpHeight можливо регулювати силу стрибку. Також слід привернути увагу до змінної isGrounded, яка перевіряє, чи знаходиться об’єкт на поверхні та обмежує стрибки.

3.2.3 Анімації для героя

Першим кроком у створенні анімації є повернення до функції Sprite Editor, за допомогою якої ми поділяємо великий спрайт астронавта та беремо

звідти спрайти з положенням астронавта при руху та стрибках. Для створення анімацій використовуємо вбудовані функції Animation та Animator. (Рис 3.7.)

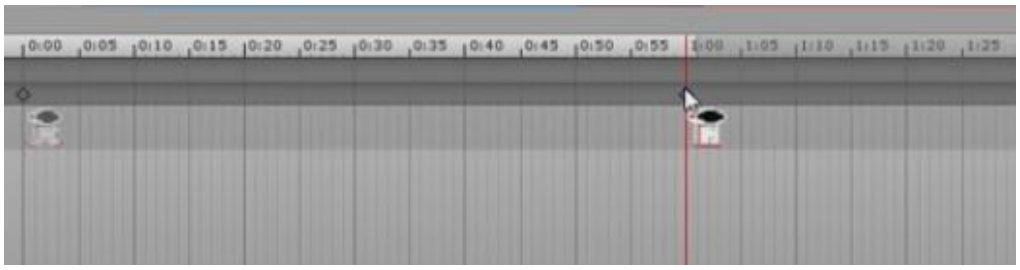


Рис 3.7 Вікно Animation.

Наступним кроком буде встановлення зв’язків кожної створеної анімації між собою, щоб забезпечити переходи від анімації однієї дії зразу до іншої. (Рис 3.8.)

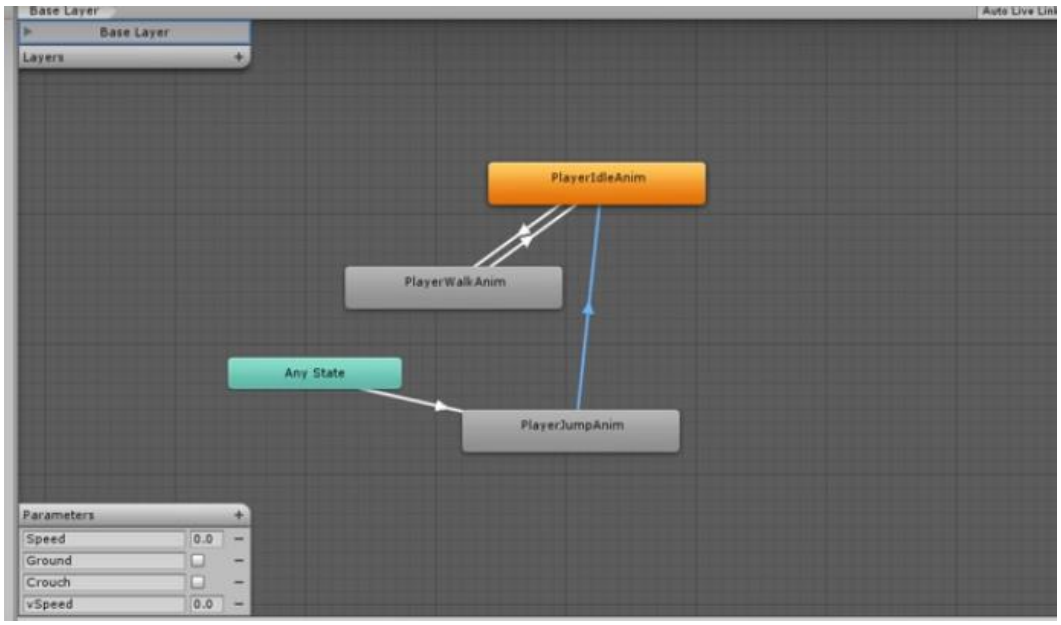


Рис 3.8 Вікно зв’язків анімацій.

3.2.4 Зброя героя

Після завершення роботи над ігровим персонажем, треба приділити увагу до способу боротьби з ворогами. На мою думку до цієї концепції підходить вогнепальна зброя, зі спрайту з астронавтом був вирізаний спрайт зі зброю та підкріплений до спрайту астронавта.

Починаючи роботу з кодом, спершу ми задаємо змінні:

```
public float fireRate = 0;
public float Damage = 10;
public LayerMask whatToHit;
```

Змінна fireRate за замовченням 0, та надає можливість налаштовувати з якою швидкістю будуть проходити постріли, змінна Damage відповідає за кількість шкоди нанесеною ворогу, а LayerMask whatToHit створює маску, за якою програма буде шукати об’єкт, до якої застосовуються постріли.

Після цього створюємо функцію з перевіркою, якщо скорострільність у дорівнює нулю, то при нажаті на кнопку буде виконуватись постріл.

```
if (fireRate == 0) {
                  if (Input.GetButtonDown ("Fire1")) {
                        Shoot();
                  }
            }
else {
                  if (Input.GetButton ("Fire1") && Time.time > timeToFire) {
                        timeToFire = Time.time + 1/fireRate;
                        Shoot();
                  }
            }
```

У іншому випадку, коли скорострільність не дорівнює нулю починається перевірка, чи була нажата кнопка пострілу з часовими інтервали.

Останньою частиною коду є синхронізація пострілу з мишкою, щоб при нажаті на кнопку постріли виконувались за слідували за курскором, а також створення лучу за допомогою методу RaycastHit2D, який служить слідом від пулі при пострілі.

```
Vector2 mousePosition = new Vector2 (Camera.main.ScreenToWorldPoint (In-
put.mousePosition).x, Camera.main.ScreenToWorldPoint(Input.mousePosition).y);
            Vector2 firePointPosition = new Vector2 (firePoint.position.x,
firePoint.position.y);
            RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePo-
sition-firePointPosition, 100, whatToHit);
```

3.2.5 Гибель та відродження героя

Перед тим, як переходити до створення ворогів, треба налаштувати умови та наслідки при смерті головного героя, такі як шкала життя та ефект відродження після смерті.

```
public class PlayerStats {
            public int Health = 100;
      }
```

Спершу створюємо клас PlayerStats та вводимо зміну Health, яка буде дорівнювати значенню 100 та буде базовим значенням очок життя персонажа.

```
DamagePlayer (int damage) {
            playerStats.Health -= damage;
            if (playerStats.Health <= 0) {
                  GameMaster.KillPlayer(this);
            }
```

Створюємо функцію DamagePlayer та створюємо умову, що при зменшенні змінної Health до менше або дорівнює 0, програма вирішує, що персонаж помер.

```
      if (transform.position.y <= fallBoundary)
                  DamagePlayer (9999999);
```

Вводимо ще одну змінну fallBoundary та створюємо умову, що якщо значення об’єкта (персонажа) по осі y буде менше ніж змінна fallBoundary, програма рахує, що персонаж впав та отримав велику кількість шкоди, що призводить до смерті героя.

```
public int fallBoundary = -20;
if (transform.position.y <= fallBoundary)

                  DamagePlayer (9999999);
```

Після створення умов для смерті героя, потрібно створити також для його відродження.

```
        public Transform playerPrefab;
      public Transform spawnPoint;
      public int spawnDelay = 2;
```

За допомогою компоненту Tranfsorm вводимо змінні playerPrefab та spawnPoint, які будут відповідати за створення копії об’єкта після його смерті та координат у яких буде з’являтись об’єкт.

Створюємо функцію KillPlayer та зазначаємо у ній, якщо персонаж помирає, ігровий об’єкт видаляється та починає діяти таймер до його відродження.

```
public static void KillPlayer (Player player) {
            Destroy (player.gameObject);
            gm.StartCoroutine (gm.RespawnPlayer());
```

3.2.6 Штучний інтелект ворогів

Перед написанням коду було завантажено з мережі інтернет безкоштовний функціонал для Unity A* Pathfinding Project, який допомагає у створенні навігації та графів.

```
public Transform target;
public float updateRate = 2f;
private Seeker seeker;
private Rigidbody2D rb;
public float speed = 300f;
public ForceMode2D fMode;
```

Важливим кроком є створення ворогів. На першому етапі створюємо клас EnemyStats, який має усі ті ж функції та цілі, що й клас PlayerStats, але буде відповідати за життя ворогів.

```
public class EnemyStats {
            public int Health = 100;
      }

      public EnemyStats stats = new EnemyStats();

      public void DamageEnemy (int damage) {
            stats.Health -= damage;
            if (stats.Health <= 0)
            {
                  GameMaster.KillEnemy (this);
            }
      }
```

Спершу створюємо зміну target, якою буде позначатися ігровий персонаж, змінну updateRate яка буде визначати кількість обновлень шляху у секунду, назначаємо клас Seeker який оброблює усі виклики шляху для об'єкта, клас Rigidbody2D, що надає об'єкту фізичні властивості, а також вводимо змінну speed яка визначає швидкість руху ворога, а також компонент ForceMode2D який надає силу об'єкту.

Вводимо змінні для визначення поточного положення об'єкта та для визначення дистанції до цілі.

```
public float nextWaypointDistance = 3;

private int currentWaypoint = 0;
```

Далі у функції старт звертаємося до компонентів Seeker та Rigitbody2D для надання об'єту фізичних властивостей, а також створюємо умову, якщо об'єкт не може знайти свою ціль, то у консолі буде виникати помилка.

```
void Start () {

            seeker = GetComponent<Seeker>();
            rb = GetComponent<Rigidbody2D>();

            if (target == null) {
                  Debug.LogError ("No Player found? PANIC!");
                  return;
            }
```

Використовуючи метод AddForce наділяємо об'єкт силою гравітації та змушуємо об'єкт рухатись до персонажа.

```
rb.AddForce (dir, fMode);

            float dist = Vector3.Distance (transform.position, path.vector-
Path[currentWaypoint]);
            if (dist < nextWaypointDistance) {
                  currentWaypoint++;
                  return;
            }
```

3.2.7 Графічний інтерфейс для героя та ворога

Наступним етапом розробки буде налаштування першого графічного інтерфейсу. Спочатку розроблюємо смугу с життям для ворогів та для ігрового

персонажа. (Рис 3.9.) За допомогою функціоналу Unity створимо пустий графічний об’єкт, до якого вже буде написано функціонал, що буде працювати с графічним інтерфейсом:

Рис 3.9 Смуга інтерфейсу життя ворога.

Створюємо функцію SetHealth, яка буде відповідати за полоску життя ігрових об’єктів. Звертаючись до методу healthbarRect.localScale визначаємо значення змінної value, яка є показником поточного життя об’єкту, та за допомогою методу heathText.text виводимо текстову інформацію с приводу поточного життя на екран.

```
public void SetHealth(int _cur, int _max)
      {
            float _value = (float)_cur / _max;

            healthBarRect.localScale = new Vector3(_value, healthBarRect.lo-
calScale.y, healthBarRect.localScale.z);
            healthText.text = _cur + "/" + _max + " HP";
      }
```

Копіюємо у вікні сцени створений скрипт до об’єкту гравця, та маємо такий саме результат графічном інтерфейсом на ігровому персонажі. (Рис 3.10.)

Рис 3.10 Смуга інтерфейсу життя героя.

3.2.8 Взаємодія між об’єктами

Після створення графічного інтерфейсу життя для ігрового героя та для ворогу, переходимо до взаємодії між ними. У нашому випадку взаємодією двух об’єктів є нанесення шкоди один одному. Почнемо з ігрового героя.

Вводимо змінну:

```
public int Damage = 10;
```

Ця змінна буде позначати, скільки шкоди наносить один постріл героя по ворогу.

Далі присвоюємо за допомогою компонента hit.collider.GetComponent об’єкту enemy можливість отримувати шкоду від пуль, а також створюємо умову, якщо пуля потрапляє по ворогу, виводити у консолі, що ворог отримав якусь кількість шкоди.

```
Enemy enemy = hit.collider.GetComponent<Enemy>();
                  if (enemy != null) {
                        enemy.DamageEnemy (Damage);
                        Debug.Log ("We hit " + hit.collider.name + " and did
" + Damage + " damage.");
                  }
```

Основні дії зі сторони героя прописані, переходимо до ворога.

Створюємо функцію, яка зчитує змінну damage та віднімає від поточного життя ворога величину змінної damage. Далі створюємо умову, якщо значення поточного життя ворога менше або дорівнює нулю, то ворог зчитується програмою як переможений гравцем та зникає. Також не забуваємо про графічній інтерфейс та передаємо за допомогою методу statusIndicator.SetHealth поточне значення життя, щоб значення виводилось на екран.

```
public void DamagePlayer (int damage) {
            stats.curHealth -= damage;
            if (stats.curHealth <= 0)
            {
                  GameMaster.KillPlayer(this);
            }

            statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
      }
```

Ігровий герой також має отримувати якусь шкоду, тому аналогічну функцію створюємо для ворогів:

```
public void DamageEnemy (int damage) {
            stats.curHealth -= damage;
            if (stats.curHealth <= 0)
            {
                  GameMaster.KillEnemy (this);
            }

            if (statusIndicator != null)
            {
                  statusIndicator.SetHealth(stats.curHealth,
stats.maxHealth);
            }
```

Створена гра передбачає, що герой отримуватиме шкоду при зіткненні з ворогом, але з кількістю ворогів яка планується у грі для надання ще одної ігрової механіки ми створимо умову, при якій у разі зіткнення героя з ворогом, останній буде знешкоджуватись після нанесення шкоди герою. Для цього створимо наступну функцію:

```
void OnCollisionEnter2D(Collision2D _colInfo)
      {
            Player _player = _colInfo.collider.GetComponent<Player>();
            if (_player != null)
            {
                  _player.DamagePlayer(stats.damage);
                  DamageEnemy(9999999);
            }
      }
```

При зіткненні двух колайдерів, героя та ворога відповідно, ворог буде отримувати 999999 кількість шкоди, що призводить до його знешкоджування.

3.2.9 Генерація хвиль ворогів

Наступним етапом розробки буде створення декількох хвиль ворогів які будуть з’являтись з певною періодичністю для надання мети та динаміки ігрового процессу.

Спершу створюємо на сцені декілька пустих ігрових об'єктів, до яких буде застосовуватись написаний скрипт. Для початку вводимо змінні, які будуть позначати кількість хвиль та періодичність їх появи:

```
 public string name;
public Transform enemy;
public int count;
public float rate;
```

Створюємо масив, який буде рахувати кількість хвиль:

```
public Wave[] waves;
 private int nextWave = 0;
 public int NextWave
 {
       get { return nextWave + 1; }
 }
```

Функціонал скрипта може мати три різни стани: SPAWNING, WAITING, COUNTING, які відповідають за появу, очікування наступної хвилі та рахунок до наступної хвилі відповідно.

Створюємо функцію SpawnEnemy, яка виконується для створених раніше на сцені пустих ігрових об’єктів та відповідає за появу ворогів при хвилях у координатах цих об’єктів.

```
void SpawnEnemy(Transform _enemy)
      {
            Debug.Log("Spawning Enemy: " + _enemy.name);

            Transform _sp = spawnPoints[ Random.Range (0, spawnPoints.Length)
];
            Instantiate(_enemy, _sp.position, _sp.rotation);
      }
```

Створюємо функцію для перевірки чи присутні живі вороги:

```
bool EnemyIsAlive()
      {
            searchCountdown -= Time.deltaTime;
            if (searchCountdown <= 0f)
            {
                  searchCountdown = 1f;
                  if (GameObject.FindGameObjectWithTag("Enemy") == null)
                  {
                        return false;
                  }
```

```
            }
            return true;
      }
```

Якщо таймер для відрахування часу до наступної хвилі менше або дорівнює 0 то переходимо у стан SPAWNING, який викликає появу нової хвилі.

```
if (waveCountdown <= 0)
            {
                  if (state != SpawnState.SPAWNING)
                  {
                        StartCoroutine( SpawnWave ( waves[nextWave] ) );
                  }
            }
```

Поява нової хвилі викликає функцію SpawnWave та переводить скрипт у режим очікування WAITING:

```
IEnumerator SpawnWave(Wave _wave)
      {
            Debug.Log("Spawning Wave: " + _wave.name);
            state = SpawnState.SPAWNING;

            for (int i = 0; i < _wave.count; i++)
            {
                  SpawnEnemy(_wave.enemy);
                  yield return new WaitForSeconds( 1f/_wave.rate );
            }

            state = SpawnState.WAITING;

            yield break;
      }
```

Кожну секунду у методі Update() виконуємо перевірку, якщо стан хвилі в очікуванні, переходимо до нескінченної перевірки, якщо усі вороги переможені, то активується функція WaveCompleted().

```
if (state == SpawnState.WAITING)
            {
                  if (!EnemyIsAlive())
                  {
                        WaveCompleted();
                  }
                  else
                  {
                        return;
                  }
            }
```

Далі створюється функція для перевірки, якщо ця хвиля була останньою, то значення змінної nextWave буде дорівнювати 0 та нові хвилі не будуть з’являтись, або починається відрахунок до наступу нової хвилі ворогів:

```
void WaveCompleted()
    {
        Debug.Log("Wave Completed!");

        state = SpawnState.COUNTING;
        waveCountdown = timeBetweenWaves;

        if (nextWave + 1 > waves.Length - 1)
        {
            nextWave = 0;
            Debug.Log("ALL WAVES COMPLETE! Looping...");
        }
        else
        {
            nextWave++;
        }
    }
```

3.2.10 Графічний інтерфейс до генератору хвиль

Розробимо графічний інтерфейс до нашого генератора хвиль. Створюємо 3 окремих текстових об’єкта для відображення номеру поточної хвилі. (Рис 3.11.)

Рис 3.11 Відображення поточної хвилі.

У методі Update() прописуємо функцію яка кожну секунду перевіряє у якому стані знаходиться скрипт WaveSpawner, який відповідає за появу

хвиль, та викликає функції UpdateCountingUI() та UpdateSwapningUI() в залежності від стану:

```
void Update () {
            switch (spawner.State)
            {
                  case WaveSpawner.SpawnState.COUNTING:
                        UpdateCountingUI();
                        break;
                  case WaveSpawner.SpawnState.SPAWNING:
                        UpdateSpawningUI();
                        break;
        }

            previousState = spawner.State;
      }
```

Функція UpdateCountingUI() відповідає за графічний інтерфейс таймеру виклику нової хвилі:

```
      void UpdateCountingUI ()
      {
            if (previousState != WaveSpawner.SpawnState.COUNTING)
            {
                  waveAnimator.SetBool("WaveIncoming", false);
                  waveAnimator.SetBool("WaveCountdown", true);
                  //Debug.Log("COUNTING");
            }
            waveCountdownText.text = ((int)spawner.WaveCountdown).ToString();
      }
```

Функція Update SpawningUI() відповідає за графічний інтерфейс повідомлення про появу нової хвилі:

```
void UpdateSpawningUI()
      {
            if (previousState != WaveSpawner.SpawnState.SPAWNING)
            {
                  waveAnimator.SetBool("WaveCountdown", false);
                  waveAnimator.SetBool("WaveIncoming", true);

                  waveCountText.text = spawner.NextWave.ToString();

                  //Debug.Log("SPAWNING");
            }
      }
```

3.2.11 Умови для завершення гри

Створимо обмеження для героя у вигляді певної кількості життів, при втраті яких гра буде завершуватись. Спершу вводимо змінну, яка буде відображати кількість життів, та графічний інтерфейс який буде відображати кількість поточних життів (3.12.)

```
private static int _remainingLives = 3;
      public static int RemainingLives
      {
            get { return _remainingLives; }
      }
```

Рис 3.12 Інтерфейс поточної кількості життів героя.

Створюємо скрипт для текстового об’єкту, який буде оновлювати кожну секунду інформацію про поточну кількість життів у героя та виводити це на екран:

```
private Text livesText;

      void Awake () {
            livesText = GetComponent<Text>();
      }

      // Update is called once per frame
      void Update () {
            livesText.text = "LIVES: " + GameMaster.Remaining-
Lives.ToString();
      }
```

Повертаємось до раніше створеного методу KillPlayer, який відповідає за знищення гравця після втрати очок життя, та дописуємо до нього нову умову: якщо значення змінної remainingLives (залишок життів) прийме значення менше або дорівнює нулю, викликається функція EndGame() про завершення гри. Інакше, герой з’являється у точці відродження.

```
public static void KillPlayer (Player player) {
```

```
            Destroy (player.gameObject);
            _remainingLives -= 1;
            if (_remainingLives <= 0)
            {
                  gm.EndGame();
            } else
            {
                  gm.StartCoroutine(gm._RespawnPlayer());
            }
      }
```

3.2.12 Меню початку та кінця гри

Створюємо графічний інтерфейс до завершення гри. Текстовий інтерфейс “Game Over” що переводиться як завершення гри, та 2 кнопки на вибір. Кнопка Retry відповідає за повторне проходження гри, та кнопка Quit, яка закриває вікно з грою. (Рис 3.13.)

Рис 3.13 Інтерфейс завершення гри.

Скрипт має дві прості функції, де Quit() відповідає за завершення гри, а Retry() за повторне завантаження сцени з грою.

```
public void Quit ()
      {
            Debug.Log("APPLICATION QUIT!");
            Application.Quit();
      }

      public void Retry ()
      {
            SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
       }
```

Створюємо нову сцену для головного меню. Створюємо знов дві кнопки, кнопку Play, яка буде відповідати за початок гри, та кнопку Quit, яка буде відповідати за завершення гри. (Рис 3.14.)

Рис 3.14 Кнопки головного меню.

Пишемо відповідний скрипт до головного меню, та створюємо у ньому дві прості функції:

```
	public void StartGame ()
	{
		SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
1);
	}

	public void QuitGame()
	{
		Debug.Log("WE QUIT THE GAME!");
		Application.Quit();
	}

}
```

StartGame () відповідає за перехід на сцену з самою гри, якщо гравець натискає на кнопку Play, а відповідно кнопка Quit за вихід із гри.

Висновок до розділу 3.

Даний розділ було цілком присвячено реалізації заданого темою проекту. Спершу було проаналізовані основні поняття графічної складової, такі як «тайл» та «спрайт». Реалізація прототипу починалась з графічних елементів, які використовувались у створені, а також пояснення основного функціоналу який використовувався для роботи з графічною складовою. Були представлені спрайти усіх елементів, які були використані у реалізації.

Перед написанням коду було проаналізовано функціонал компонентів ігрового рушія, за допомогою яких об’єктам надаються фізичні властивості та елементи яких використовуються на протязі усієї реалізації.

Наступний етап розділу включає у собі докладне пояснення до кожного етапу та кроку. У першу чергу працювали з моделлю героя гри, написанням скриптів для руху та стрибків, описом основних класів, функцій та методів які використовуються для надання життя герою, а також представлення схеми реалізації анімації головного героя для руху. Докладно описано створення пострілів, описано функції та методи для створення ефекту пострілу та руху слідів від зброї за курсором миші. Було докладно описано створення ворогів героя та написання штучного інтелекту для ворогів, а також умови для смерті та відродження героя. Створено простий графічний інтерфейс над моделями героя та ворогів який показує кількість очок життя відповідних об’єктів, а також детальний опис створених скриптів для реалізації цього інтерфейсу та його зміни від кількості поточного життя об’єкту.

У наступному кроці описано один з найважливіших етапів реалізації, такий як автоматизацію для створення нових хвиль ворогів. Було детально описано кожний ключовий момент та приділено увагу до найважливіших моментів при написанні скриптів. Створено графічний інтерфейс з анімацією тай-

меру відрахування часу до появи хвилі ворогів, а також інтерфейс анонсів номеру хвилі. Створено також інтерфейс у вигляді текстового надпису, який інформує гравця про поточну кількість життів.

Завершальним кроком реалізації є створення умови для закінчення гри, а також відповідного інтерфейсу. Описано створення умови для поразки або перемоги гравця. Детальний опис створення графічного інтерфейсу у вигляді меню завершення гри, на якому гравець має вибір спробувати ще раз чи завершити гру. Завершальним кроком є створення першої сцени гри, у вигляді стартового меню, яке надає гравцю вибір почати гру або вийти з неї.

РОЗДІЛ 4
ІНСТРУКЦІЯ КОРИСТУВАЧА

4.1 Стартове меню

Першим, що побачить перед собою потенційний гравець буде ігрове меню. Ігрове меню має 2 кнопки: Play та Quit. Натиснувши Play гравець переходить до гри, а натиснувши Quit закриває ігрове меню. Загальний фон та інтерфейс меню був виконаний у мінімалістичному стилі з приємним синім кольором, а також спрайтами з інопланетною істотою та платформою. (Рис 4.1.)

Рис 4.1 Стартове меню гри.

4.2 Інтерфейс та ігровий процес

Після натиснення кнопки Play гра починається. Управління у грі відбувається за допомогою кнопок на клавіатурі Space, A, D. Де A – це рух вліво, D – рух вправо, а Space – стрибок. Постріли зі зброї відбуваються за допомогою ЛКМ (лівої кнопки миші).

Гравець бачить перед собою свого ігрового героя, навколишній світ, платформи, по яким можна стрибати, відлік до початку нападу ворогів, кількість його життів та графічний інтерфейс над моделлю героя, який вказує на кількість очок життя. (Рис 4.2.)

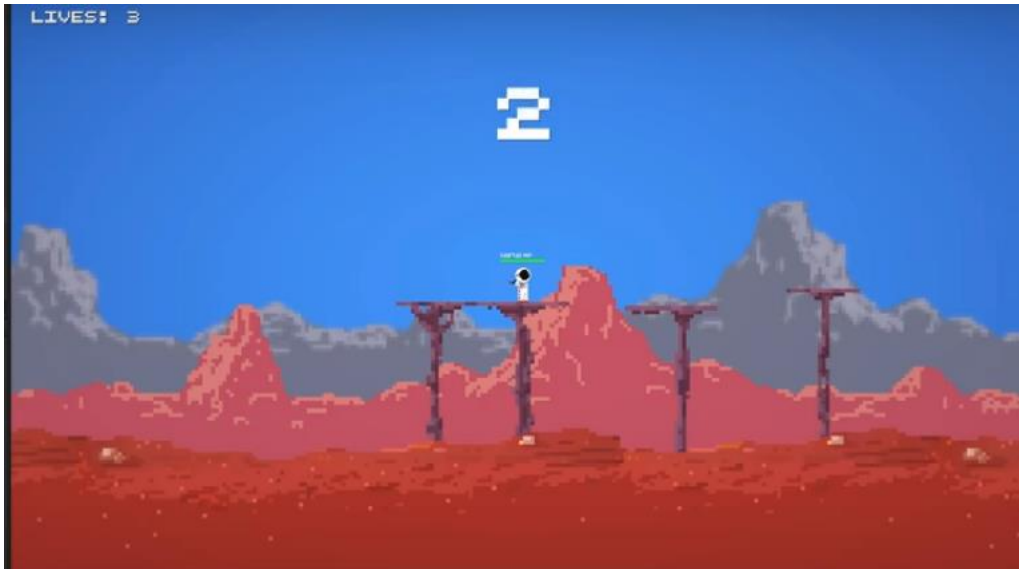


Рис 4.2 Інтерфейс гри.

Після кінця відліку хвилі ворогів, з’являється надпис про наступ першої хвилі. (Рис 4.3.)

Рис 4.3 Анонсування першої хвилі ворогів.

Після надпису про наступ першої хвилі, вороги починають рухатись до героя та намагатись нанести йому шкоду, а гравець повинен за допомогою пострілів знешкодити ворогів. (Рис 4.4.)

Рис 4.4 Постріли по ворогу.

Якщо герой був переможений при наступі ворогів, його кількість життів знизиться, а на екрані з’явиться відлік до відродження героя. (Рис 4.5.)

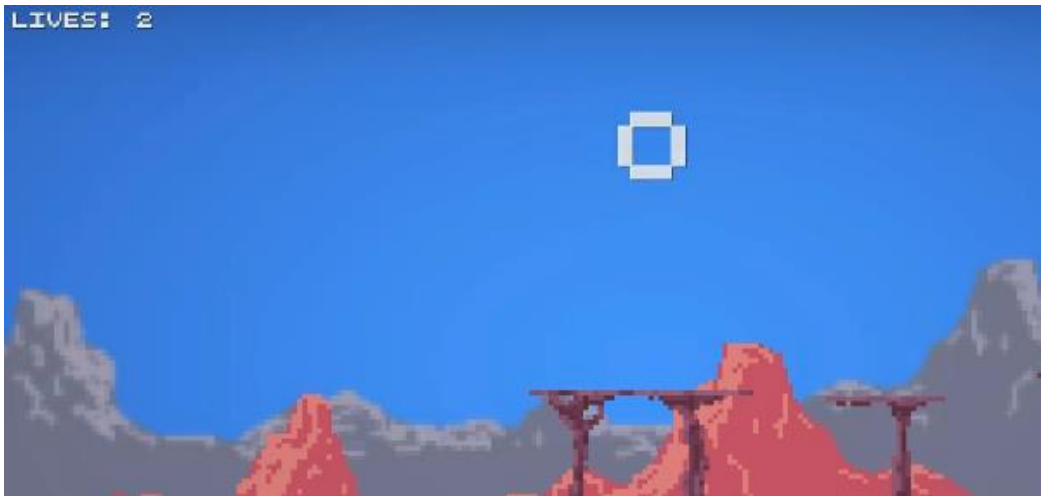


Рис 4.5 Гибель героя та відлік до відродження.

Якщо хвиля була переможена, на екрані з’являється надпис про наступ другої хвилі. (Рис 4.6.)

Рис 4.6 Інтерфейс з анонсом другої хвилі ворогів.

Оскільки жанр створюваної гри саме Платформер, то крім того, що гравець повинен захитащись від наступу ворогів, він повинен рухатись по платформах, які находяться на відстані один від одної та робити це обережно, адже падіння призводить до погибелі героя та втрати одного життя. (Рис 4.7.)

Рис 4.7 Знімок екрана з платформами.

4.3 Умови та меню завершення гри

Умовою для завершення гри є знешкодження усіх хвиль ворогів, або втрата усіх життів.

Після виконання однієї із умов для перемоги або поразки, гравець бачить перед собою меню завершення гри, де йому пропонуються вибір, спробувати ще раз, або завершити та вийти із гри. (Рис 4.8.)

Рис 4.8 Меню завершення гри.

Висновок до розділу 4.

Даний розділ бакалаврського проекту було присвячено формуванню інструкції роботи користувача з створеним прототипом гри, було докладно описано інтерфейс головного меню, можливості, якими володіє ігровий герой, такі як біг, стрибки та постріли. Описано стартовий ігровий екран та графічні елементи на ньому, такі як кількість життів, відлік до появи першої хвилі ворогів, анонсування появи першої хвилі та поява наступних. Враховуючи поставлені вимоги до нашого прототипу, було описано мету гри, умови для перемоги або програшу, можливості для боротьби с ворогами та проаналізовано наслідки після гибелі або проходження хвилі гравцем. Описано етап завершення гри гравцем, при якому він має можливість спробувати ще раз, або вийти з гри.

Виходячи з матеріалів розділу можна зробити висновок, що основним функціоналом створеного прототипу гри є:

- Стартове меню з можливістю вибору;
- Простий та зрозумілий інтерфейс;
- Вороги зі штучним інтелектом які мають мету знешкодити героя;
- Автоматизація появи нових хвиль ворогів;
- Умови для перемоги або поразки;
- Меню завершення гри з можливістю вибору.

ВИСНОВКИ

Робота складається з чотирьох розділів. Перший розділ містить загальні відомості про комп'ютерні ігри та їх жанрову класифікацію. Крім цього, було проаналізовано основні переваги та підсумовано недоліки існуючих сучасних прототипів.

Другий розділ містить у собі опис вибраного жанру проекту, а також аналіз існуючих методів і функціоналу розробки ігор, та обґрунтування вибраного ігрового рушія для розробки. Крім цього було описано повний алгорим створення гри, проаналізовано потенційну аудиторію гравців та загальну актуальність проекту, а також сформульовано мету розробки та функціонал гри, якому вона повинна дотримуватися.

У третьому розділі описана повна поетапна реалізація проекту. Спершу було визначено основні поняття стосовно візуальної складової проекту, а потім описано основні графічні компоненти та елементи які використовувались при розробці гри. Крім цього було детально описано кожен етап розробки з докладним описом кожного важливого методу та класу, який використовувався при написанні коду. Також було описано створення меню початку та завершення гри.

Четвертий розділ є найкоротшим та представляє собою просту інструкцію користувача, або потенційного гравця. Докладно описано кожен крок, який повинен виконати гравець, а також кожен елемент інтерфейсу. Описана бойова система, мета, умови для перемоги чи програшу, та робота з меню початком і кінцем гри.

Завершаючим результатом проектної роботи є робочий прототип гри у жанрі Платформер, який не має високих технічних вимог, має просте меню, управління, інтерфейс, мету гри, а також умови для її завершення. Даний прототип гри може бути доповнений новими деталями і елементами та офіційно випущений як гра на базі якогось інтернет-ресурсу.

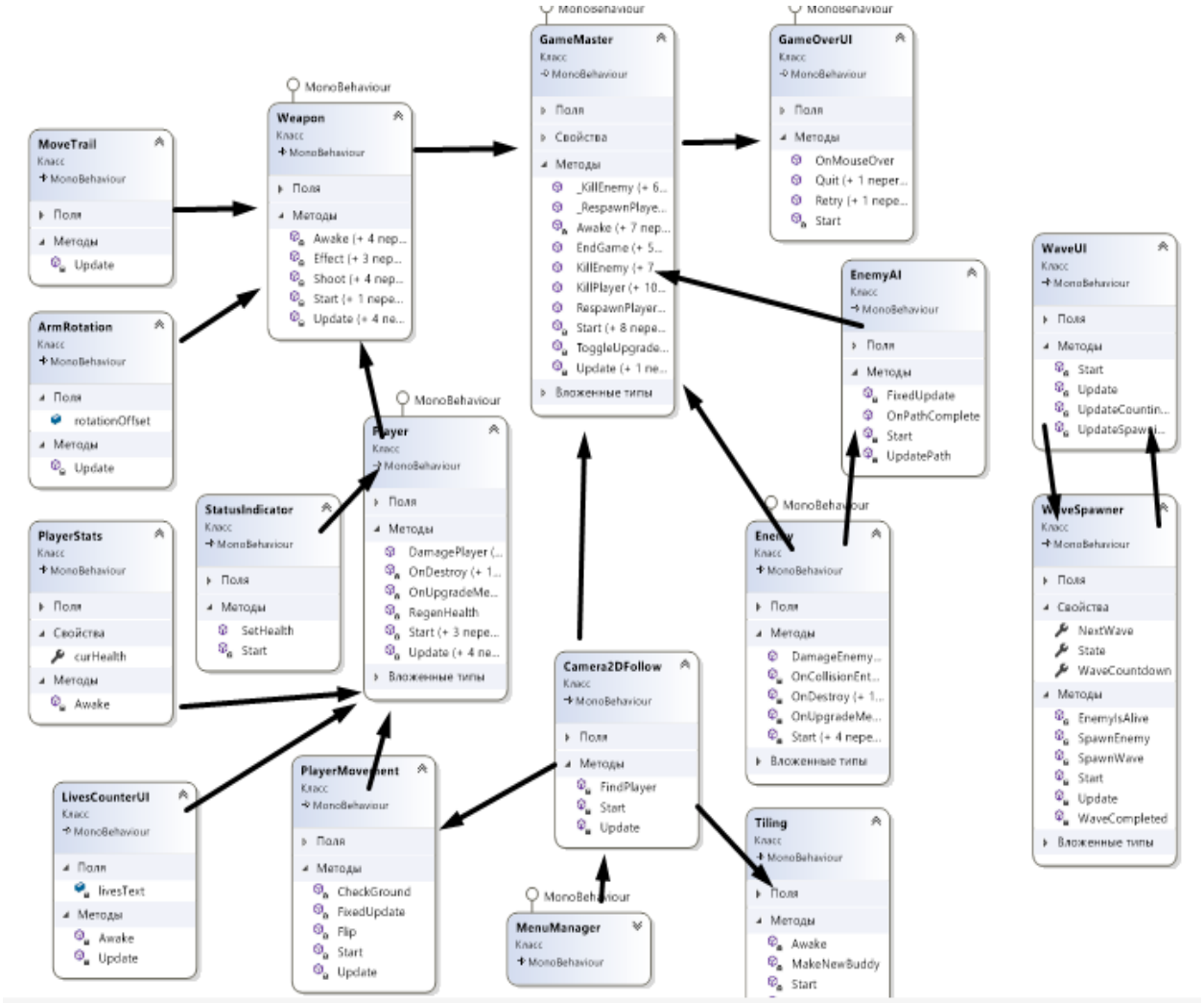
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ:

1. Комп'ютерна гра [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/PC_game.
2. Жанр Платформер [Електронний ресурс] – Режим доступу до ресурсу: en.wikipedia.org/wiki/Platform_game.
3. Класифікація ігор за жанрами [Електронний ресурс] – Режим доступу до ресурсу: https://gamesisart.ru/.
4. Гра Mark of the Ninja [Електронний ресурс] – Режим доступу до ресурсу: https://www.klei.com/games/mark-ninja.
5. Гра Cuphead [Електронний ресурс] – Режим доступу до ресурсу: https://www.xbox.com/ru-RU/games/cuphead.
6. Гра Starbound [Електронний ресурс] – Режим доступу до ресурсу: https://playstarbound.com/.
7. Гра Dead Cells [Електронний ресурс] – Режим доступу до ресурсу: https://deadcells.gamepedia.com/Dead_Cells_Wiki.
8. Найкращі ігрові рушії [Електронний ресурс] – Режим доступу до ресурсу: https://www.gamedesigning.org/career/video-game-engines.
9. Unreal Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.unrealengine.com/.
10. Cry Engine [Електронний ресурс] – Режим доступу до ресурсу: https://www.cryengine.com/.
11. Unity Game Engine [Електронний ресурс] – Режим доступу до ресурсу: https://unity.com/ru.
12. Unity Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/index.html.
13. David B. Hands-On Game Development Patterns with Unity / Baron David., 2019. – 116 с.

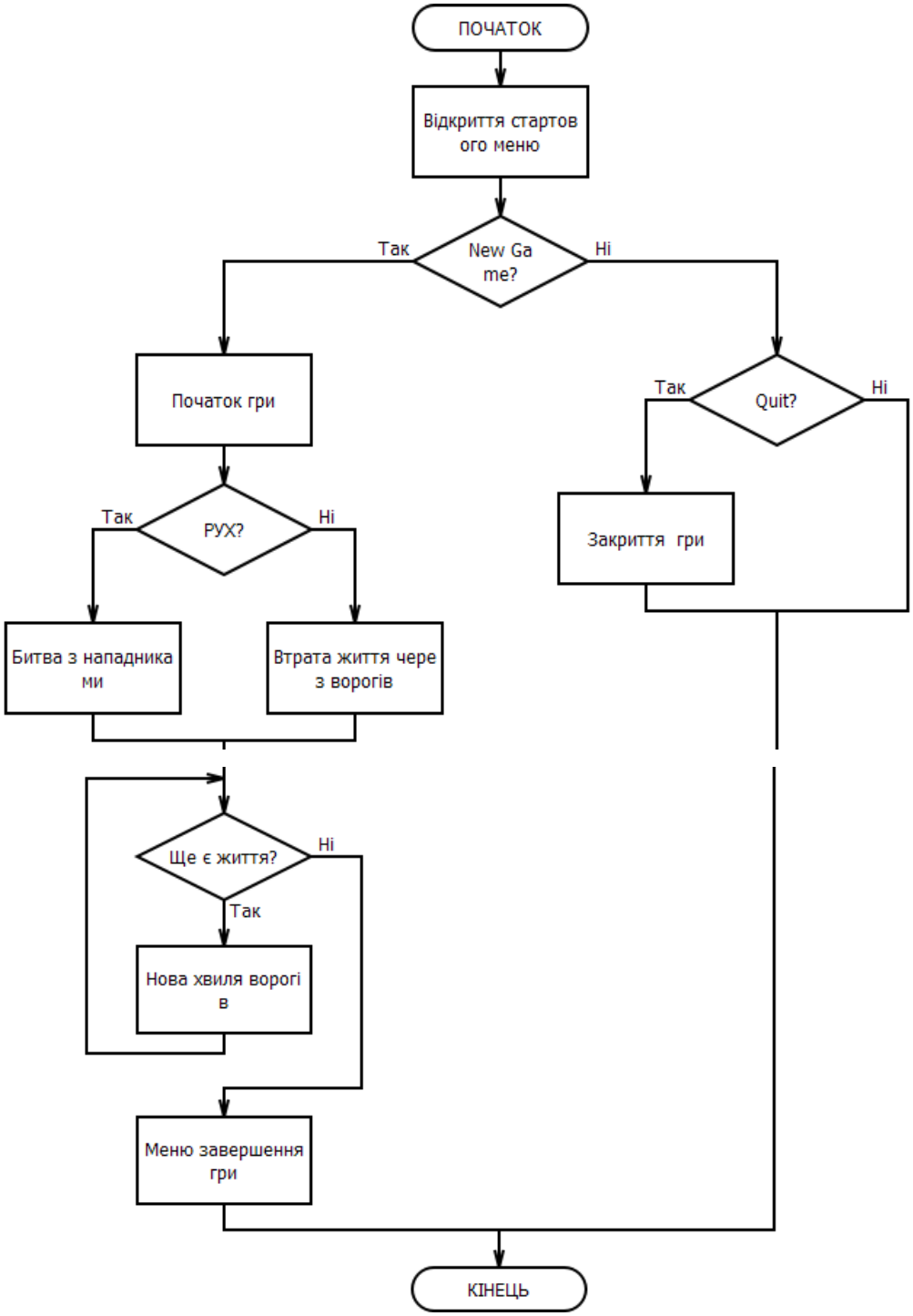
14. Створення 2D на Unity [Електронний ресурс] – Режим доступу до ресурсу: https://skillbox.ru/media/code/kak_sozdat_prostuyu_2d_igru_na_unity/.
15. 2D на Unity, докладний посібник [Електронний ресурс] – Режим доступу до ресурсу: http://websketches.ru/blog/2d-igra-na-unity-podrobnoye-rukovodstvo-p1.
16. Пояснення про спрайти, тайли [Електронний ресурс] – Режим доступу до ресурсу: https://www.econdude.pw/2017/08/chto-takoe-graficheskij-sprajt-sprite.html.
17. Hocking J. Unity in Action. Multiplatform game development in C# with Unity 5 / Joseph Hocking., 2015. – 352 с.
18. Smith G. Basic Math for Game Development with Unity 3D / G. Smith, K. Sung., 2018. – 279 с.
19. Thorn A. Mastering Unity Scripting / Alan Thorn., 2015. – 380 с.
20. Ferrone H. Mastering Unity Scripting / Harrison Ferrone., 2016. – 379 с.
21. Ferrone H. Learning C# by Developing Games with Unity 2019: Code in C# and build 3D games with Unity, 4th Edition / Harrison Ferrone., 2019. – 342 с.
22. Halpern J. Developing 2D Games with Unity: Independent Game Programming with C# / Jared Halpern., 2018. – 408 с.

ДОДАТОК А

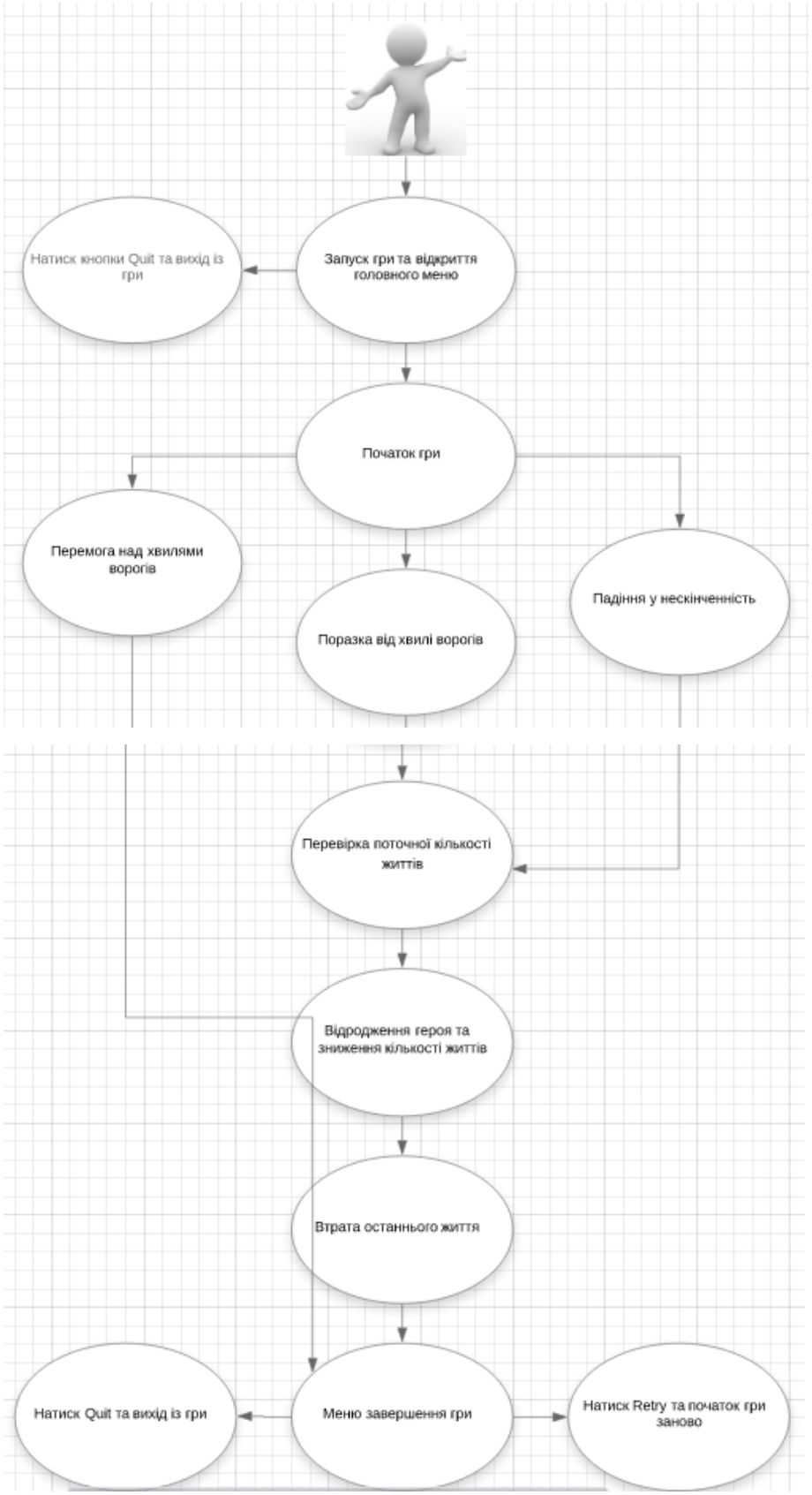
ГРАФІЧНІ МАТЕРІАЛИ



					ІАЛЦ.622500.008 А2	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Ставцев Д. І.			Гра у жанрі Платформер Функціональна схема	Літ. / Аркуш / Аркушів 1 / 1
Перевірив		Жабін В. І.				
Реценз.						НТУУ «КПІ», ФІОТ, ІО-62
Н. Контр.		Сімоненко В.П.				
Затв.						



					ІАЛЦ.622500.006 А1	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Ставцев Д. І.			Гра у жанрі Платформер Принципова схема алгоритму	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Жабін В. І.				
Реценз.						НТУУ «КПІ», ФІОТ, ІО-62
Н. Контр.		Сімоненко В.П.				
Затв.						



					ІАЛЦ.622500.002 ТЗ	
Зм.	*Арк.*	*№ докум.*	*Підпис*	*Дата*		
Розробив		*Ставцев Д. І.*			*Гра у жанрі Платформер*	*Літ.* / *Аркуш* / *Аркушів*
Перевірив		*Жабін В. І.*				*1* / *1*
Реценз.					*Структурна схема*	
Н. Контр.		*Сімоненко В.П.*				*НТУУ «КПІ», ФІОТ, ІО-62*
Затв.						

ДОДАТОК Б
ЛІСТИНГ ПРОГРАМИ

Файл PlayerMovement.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEditor.Build;
using UnityEngine;

public class PlayerMovement : MonoBehaviour


{
   Rigidbody2D rb;
   public float speed;
   public float jumpHeight;
   public Transform groundCheck;
   bool isGrounded;


   // Start is called before the first frame update
   void Start()
   {
      rb = GetComponent<Rigidbody2D>();

   }

   // Update is called once per frame
   void Update()
   {
      Flip();
      CheckGround();
   }

   void FixedUpdate()
   {
      rb.velocity = new Vector2(Input.GetAxis("Horizontal") * speed, rb.velocity.y);
      if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
         rb.AddForce(transform.up * jumpHeight, ForceMode2D.Impulse);
   }

   void Flip()
   {
      if (Input.GetAxis("Horizontal") > 0)
         transform.localRotation = Quaternion.Euler(0, 0, 0);
      if (Input.GetAxis("Horizontal") < 0)
```

```
            transform.localRotation = Quaternion.Euler(0, 180, 0);
        }

        void CheckGround()
        {
            Collider2D[] colliders = Physics2D.OverlapCircleAll(groundCheck.position, 0.2f);
            isGrounded = colliders.Length > 1;



}
```

Файл Camera2DFollow.cs

```
using System;
using UnityEngine;

namespace UnityStandardAssets._2D
{
    public class Camera2DFollow : MonoBehaviour
    {
        public Transform target;
        public float damping = 1;
        public float lookAheadFactor = 3;
        public float lookAheadReturnSpeed = 0.5f;
        public float lookAheadMoveThreshold = 0.1f;

        private float m_OffsetZ;
        private Vector3 m_LastTargetPosition;
        private Vector3 m_CurrentVelocity;
        private Vector3 m_LookAheadPos;

        // Use this for initialization
        private void Start()
        {
            m_LastTargetPosition = target.position;
            m_OffsetZ = (transform.position - target.position).z;
            transform.parent = null;
        }


        // Update is called once per frame
        private void Update()
        {
            // only update lookahead pos if accelerating or changed direction
            float xMoveDelta = (target.position - m_LastTargetPosition).x;

            bool updateLookAheadTarget = Mathf.Abs(xMoveDelta) > lookAhead-
MoveThreshold;

            if (updateLookAheadTarget)
            {
                m_LookAheadPos = lookAheadFactor*Vec-
tor3.right*Mathf.Sign(xMoveDelta);
```

```
            }
            else
            {
               m_LookAheadPos = Vector3.MoveTowards(m_LookAheadPos, Vector3.zero,
        Time.deltaTime*lookAheadReturnSpeed);
            }

            Vector3 aheadTargetPos = target.position + m_LookAheadPos + Vector3.for-
        ward*m_OffsetZ;

          Vector3 newPos = Vector3.SmoothDamp(transform.position, aheadTargetPos, ref
        m_CurrentVelocity, damping);

            transform.position = newPos;

            m_LastTargetPosition = target.position;
         }
       }
}
```

Файл Parallaxing.cs

```
using UnityEngine;
using System.Collections;

public class Parallaxing : MonoBehaviour {

        public Transform[] backgrounds;                 // Array (list) of all the back-
and foregrounds to be parallaxed
        private float[] parallaxScales;                 // The proportion of the camera's
movement to move the backgrounds by
        public float smoothing = 1f;                    // How smooth the parallax is going
to be. Make sure to set this above 0

        private Transform cam;                                  // reference to the
main cameras transform
        private Vector3 previousCamPos;                 // the position of the camera
in the previous frame

        // Is called before Start(). Great for references.
        void Awake () {
                // set up camera the reference
                cam = Camera.main.transform;
        }

        // Use this for initialization
        void Start () {
                // The previous frame had the current frame's camera position
                previousCamPos = cam.position;

                // asigning coresponding parallaxScales
                parallaxScales = new float[backgrounds.Length];
```

```
                for (int i = 0; i < backgrounds.Length; i++) {
                        parallaxScales[i] = backgrounds[i].position.z*-1;
                }
        }

        // Update is called once per frame
        void Update () {

                // for each background
                for (int i = 0; i < backgrounds.Length; i++) {
                        // the parallax is the opposite of the camera movement because
the previous frame multiplied by the scale
                        float parallax = (previousCamPos.x - cam.position.x) *
parallaxScales[i];

                        // set a target x position which is the current position plus the
parallax
                        float backgroundTargetPosX = backgrounds[i].position.x + parallax;

                        // create a target position which is the background's current
position with it's target x position
                        Vector3 backgroundTargetPos = new Vector3
(backgroundTargetPosX, backgrounds[i].position.y, backgrounds[i].position.z);

                        // fade between current position and the target position using lerp
                        backgrounds[i].position = Vector3.Lerp (backgrounds[i].position,
backgroundTargetPos, smoothing * Time.deltaTime);
                }

                // set the previousCamPos to the camera's position at the end of the
frame
                previousCamPos = cam.position;
        }
}
```

Файл Tiling.cs

```
using UnityEngine;
using System.Collections;

[RequireComponent (typeof(SpriteRenderer))]

public class Tiling : MonoBehaviour {

        public int offsetX = 2;                         // the offset so that we don't get any
weird errors

        // these are used for checking if we need to instantiate stuff
        public bool hasARightBuddy = false;
        public bool hasALeftBuddy = false;
```

```
	public bool reverseScale = false;	// used if the object is not tilable

	private float spriteWidth = 0f;			// the width of our element
	private Camera cam;
	private Transform myTransform;

	void Awake () {
		cam = Camera.main;
		myTransform = transform;
	}

	// Use this for initialization
	void Start () {
		SpriteRenderer sRenderer = GetComponent<SpriteRenderer>();
		spriteWidth = sRenderer.sprite.bounds.size.x;
	}

	// Update is called once per frame
	void Update () {
		// does it still need buddies? If not do nothing
		if (hasALeftBuddy == false || hasARightBuddy == false) {
			// calculate the cameras extend (half the width) of what the camera can see in world coordinates
			float camHorizontalExtend = cam.orthographicSize * Screen.width/Screen.height;

			// calculate the x position where the camera can see the edge of the sprite (element)
			float edgeVisiblePositionRight = (myTransform.position.x + spriteWidth/2) - camHorizontalExtend;
			float edgeVisiblePositionLeft = (myTransform.position.x - spriteWidth/2) + camHorizontalExtend;

			// checking if we can see the edge of the element and then calling MakeNewBuddy if we can
			if (cam.transform.position.x >= edgeVisiblePositionRight - offsetX && hasARightBuddy == false)
			{
				MakeNewBuddy (1);
				hasARightBuddy = true;
			}
			else if (cam.transform.position.x <= edgeVisiblePositionLeft + offsetX && hasALeftBuddy == false)
			{
				MakeNewBuddy (-1);
				hasALeftBuddy = true;
			}
		}
	}

	// a function that creates a buddy on the side required
	void MakeNewBuddy (int rightOrLeft) {
```

```
                // calculating the new position for our new buddy
                Vector3 newPosition = new Vector3(myTransform.position.x + myTrans-
        form.localScale.x * spriteWidth * rightOrLeft, myTransform.position.y, myTrans-
        form.position.z);
                // instantating our new body and storing him in a variable
                Transform newBuddy = Instantiate (myTransform, newPosition, myTrans-
        form.rotation) as Transform;

                // if not tilable let's reverse the x size og our object to get rid of ugly seams
                if (reverseScale == true) {
                        newBuddy.localScale = new Vector3 (newBuddy.localScale.x*-1,
        newBuddy.localScale.y, newBuddy.localScale.z);
                }

                newBuddy.parent = myTransform;
                if (rightOrLeft > 0) {
                        newBuddy.GetComponent<Tiling>().hasALeftBuddy = true;
                }
                else {
                        newBuddy.GetComponent<Tiling>().hasARightBuddy = true;
                }
        }
}
```

Файл Movetrail.cs

```
using UnityEngine;
using System.Collections;

public class MoveTrail : MonoBehaviour {

        public int moveSpeed = 230;

        // Update is called once per frame
        void Update () {
                transform.Translate (Vector3.right * Time.deltaTime * moveSpeed);
                Destroy (gameObject, 1);
        }
}
```

Файл ArmRotation.cs

```
using UnityEngine;
using System.Collections;

public class ArmRotation : MonoBehaviour {

        public int rotationOffset = 90;

        // Update is called once per frame
        void Update () {
                // subtracting the position of the player from the mouse position
                Vector3 difference = Camera.main.ScreenToWorldPoint (Input.mousePo-
        sition) - transform.position;
                difference.Normalize ();                // normalizing the vector. Meaning
        that all the sum of the vector will be equal to 1
```

```
                float rotZ = Mathf.Atan2 (difference.y, difference.x) * Mathf.Rad2Deg;   //
        find the angle in degrees
                transform.rotation = Quaternion.Euler (0f, 0f, rotZ + rotationOffset);
        }
}
```

Файл GameMaster.cs

```
using UnityEngine;
using System.Collections;
using UnityStandardAssets.ImageEffects;

public class GameMaster : MonoBehaviour
{
        public static GameMaster gm;

        [SerializeField]
        private int maxLives = 3;
        private static int _remainingLives;
        public static int RemainingLives
        {
                get { return _remainingLives; }
        }

        [SerializeField]
        private int startingMoney;
        public static int Money;

        void Awake()
        {
                if (gm == null)
                {
                        gm = GameObject.FindGameObjectWithTag("GM").GetCompo-
        nent<GameMaster>();
                }
        }

        public Transform playerPrefab;
        public Transform spawnPoint;
        public float spawnDelay = 2;
        public Transform spawnPrefab;
        public string respawnCountdownSoundName = "RespawnCountdown";
        public string spawnSoundName = "Spawn";

        public string gameOverSoundName = "GameOver";

        public CameraShake cameraShake;

        [SerializeField]
        private GameObject gameOverUI;

        [SerializeField]
```

```
	private GameObject upgradeMenu;

	[SerializeField]
	private WaveSpawner waveSpawner;

	public delegate void UpgradeMenuCallback(bool active);
	public UpgradeMenuCallback onToggleUpgradeMenu;

	//cache
	private AudioManager audioManager;

	void Start()
	{
		if (cameraShake == null)
		{
			Debug.LogError("No camera shake referenced in GameMaster");
		}

		_remainingLives = maxLives;

		Money = startingMoney;

		//caching
		audioManager = AudioManager.instance;
		if (audioManager == null)
		{
			Debug.LogError("FREAK OUT! No AudioManager found in the
scene.");
		}
	}

	void Update()
	{
		if (Input.GetKeyDown(KeyCode.U))
		{
			ToggleUpgradeMenu();
		}
	}

	private void ToggleUpgradeMenu()
	{
		upgradeMenu.SetActive(!upgradeMenu.activeSelf);
		waveSpawner.enabled = !upgradeMenu.activeSelf;
	onToggleUpgradeMenu.Invoke(upgradeMenu.activeSelf);
	}

	public void EndGame()
	{
		audioManager.PlaySound(gameOverSoundName);

		Debug.Log("GAME OVER");
		gameOverUI.SetActive(true);
```

```
	}

	public IEnumerator _RespawnPlayer()
	{
		audioManager.PlaySound(respawnCountdownSoundName);
		yield return new WaitForSeconds(spawnDelay);

		audioManager.PlaySound(spawnSoundName);
		Instantiate(playerPrefab, spawnPoint.position, spawnPoint.rotation);
		GameObject clone = Instantiate(spawnPrefab, spawnPoint.position,
spawnPoint.rotation) as GameObject;
		Destroy(clone, 3f);
	}

	public static void KillPlayer(Player player)
	{
		Destroy(player.gameObject);
		_remainingLives -= 1;
		if (_remainingLives <= 0)
		{
			gm.EndGame();
		}
		else
		{
			gm.StartCoroutine(gm._RespawnPlayer());
		}
	}

	public static void KillEnemy(Enemy enemy)
	{
		gm._KillEnemy(enemy);
	}
	public void _KillEnemy(Enemy _enemy)
	{
		// Let's play some sound
		audioManager.PlaySound(_enemy.deathSoundName);

		// Gain some money
		Money += _enemy.moneyDrop;
		audioManager.PlaySound("Money");

		// Add particles
		GameObject _clone = Instantiate(_enemy.deathParticles, _enemy.trans-
form.position, Quaternion.identity) as GameObject;
		Destroy(_clone, 5f);

		// Go camerashake
		cameraShake.Shake(_enemy.shakeAmt, _enemy.shakeLength);
		Destroy(_enemy.gameObject);
	}

}
```

Файл Enemy.cs

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(EnemyAI))]
public class Enemy : MonoBehaviour {

	[System.Serializable]
	public class EnemyStats {
		public int maxHealth = 100;

		private int _curHealth;
		public int curHealth
		{
			get { return _curHealth; }
			set { _curHealth = Mathf.Clamp (value, 0, maxHealth); }
		}

		public int damage = 40;

		public void Init()
		{
			curHealth = maxHealth;
		}
	}

	public EnemyStats stats = new EnemyStats();

	public Transform deathParticles;

	public float shakeAmt = 0.1f;
	public float shakeLength = 0.1f;

	public string deathSoundName = "Explosion";

	public int moneyDrop = 10;

	[Header("Optional: ")]
	[SerializeField]
	private StatusIndicator statusIndicator;

	void Start()
	{
		stats.Init();

		if (statusIndicator != null)
		{
			statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
		}

		GameMaster.gm.onToggleUpgradeMenu += OnUpgradeMenuToggle;
```

```
            if (deathParticles == null)
            {
                Debug.LogError("No death particles referenced on Enemy");
            }
        }

        void OnUpgradeMenuToggle(bool active)
        {
            GetComponent<EnemyAI>().enabled = !active;
        }

        public void DamageEnemy (int damage) {
            stats.curHealth -= damage;
            if (stats.curHealth <= 0)
            {
                GameMaster.KillEnemy (this);
            }

            if (statusIndicator != null)
            {
                statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
            }
        }

        void OnCollisionEnter2D(Collision2D _colInfo)
        {
            Player _player = _colInfo.collider.GetComponent<Player>();
            if (_player != null)
            {
                _player.DamagePlayer(stats.damage);
                DamageEnemy(9999999);
            }
        }

        void OnDestroy ()
        {
            GameMaster.gm.onToggleUpgradeMenu -= OnUpgradeMenuToggle;
        }
}
```

Файл Player.cs

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Platformer2DUserControl))]
public class Player : MonoBehaviour
{
    public int fallBoundary = -20;

    public string deathSoundName = "DeathVoice";
    public string damageSoundName = "Grunt";

    private AudioManager audioManager;
```

```
	[SerializeField]
	private StatusIndicator statusIndicator;

	private PlayerStats stats;

	void Start()
	{
		stats = PlayerStats.instance;

		stats.curHealth = stats.maxHealth;

		if (statusIndicator == null)
		{
			Debug.LogError("No status indicator referenced on Player");
		}
		else
		{
			statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
		}

		GameMaster.gm.onToggleUpgradeMenu += OnUpgradeMenuToggle;

		audioManager = AudioManager.instance;
		if (audioManager == null)
		{
			Debug.LogError("PANIC! No audiomanager in scene.");
		}

	InvokeRepeating("RegenHealth", 1f/stats.healthRegenRate, 1f/stats.healthRe-
genRate);
	}

	void RegenHealth ()
	{
		stats.curHealth += 1;
		statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
	}

	void Update()
	{
		if (transform.position.y <= fallBoundary)
			DamagePlayer(9999999);
	}

	void OnUpgradeMenuToggle(bool active)
	{
		GetComponent<Platformer2DUserControl>().enabled = !active;
		Weapon _weapon = GetComponentInChildren<Weapon>();
		if (_weapon != null)
			_weapon.enabled = !active;
	}
```

```
	void OnDestroy()
	{
		GameMaster.gm.onToggleUpgradeMenu -= OnUpgradeMenuToggle;
	}

	public void DamagePlayer(int damage)
	{
		stats.curHealth -= damage;

		if (stats.curHealth <= 0)
		{
			//play death sound
			audioManager.PlaySound(deathSoundName);

			//kill player
			GameMaster.KillPlayer(this);
		}
		else
		{
			//play damage sound
			audioManager.PlaySound(damageSoundName);
		}

		statusIndicator.SetHealth(stats.curHealth, stats.maxHealth);
	}
```

Файл PlayeStats.cs

```
using UnityEngine;

public class PlayerStats : MonoBehaviour
{
	public static PlayerStats instance;

	public int maxHealth = 100;

	private int _curHealth;
	public int curHealth
	{
		get { return _curHealth; }
		set { _curHealth = Mathf.Clamp(value, 0, maxHealth); }
	}

	public float healthRegenRate = 2f;

	public float movementSpeed = 10f;

	void Awake()
	{
		if (instance == null)
		{
```

```
                instance = this;
            }
        }
}
```

Файл UpgradeMenu.cs

```
using UnityEngine;
using UnityEngine.UI;

public class UpgradeMenu : MonoBehaviour {

    [SerializeField]
    private Text healthText;

    [SerializeField]
    private Text speedText;

    [SerializeField]
    private float healthMultiplier = 1.3f;

    [SerializeField]
    private float movementSpeedMultiplier = 1.3f;

    [SerializeField]
    private int upgradeCost = 50;

    private PlayerStats stats;

    void OnEnable ()
    {
        stats = PlayerStats.instance;
        UpdateValues();
    }

    void UpdateValues ()
    {
        healthText.text = "HEALTH: " + stats.maxHealth.ToString();
        speedText.text = "SPEED: " + stats.movementSpeed.ToString();
    }

    public void UpgradeHealth ()
    {
        if (GameMaster.Money < upgradeCost)
        {
            AudioManager.instance.PlaySound("NoMoney");
            return;
        }

        stats.maxHealth = (int)(stats.maxHealth * healthMultiplier);

        GameMaster.Money -= upgradeCost;
        AudioManager.instance.PlaySound("Money");
```

```
                UpdateValues();
        }

        public void UpgradeSpeed()
        {
                if (GameMaster.Money < upgradeCost)
                {
                        AudioManager.instance.PlaySound("NoMoney");
                        return;
                }

                stats.movementSpeed = Mathf.Round (stats.movementSpeed * move-
    mentSpeedMultiplier);

                GameMaster.Money -= upgradeCost;
                AudioManager.instance.PlaySound("Money");

                UpdateValues();
        }

}
```

Файл MoneyCounterUI.cs

```
    using UnityEngine;
    using UnityEngine.UI;

    [RequireComponent(typeof(Text))]
    public class MoneyCounterUI : MonoBehaviour
    {

        private Text moneyText;

        void Awake()
        {
                moneyText = GetComponent<Text>();
        }

        // Update is called once per frame
        void Update()
        {
                moneyText.text = "MONEY: " + GameMaster.Money.ToString();
        }
}
```

Файл AudioManager.cs

```
    using UnityEngine;

    [System.Serializable]
    public class Sound
    {

        public string name;
        public AudioClip clip;
```

```
        [Range(0f, 1f)]
        public float volume = 0.7f;
        [Range(0.5f, 1.5f)]
        public float pitch = 1f;

        [Range(0f, 0.5f)]
        public float randomVolume = 0.1f;
        [Range(0f, 0.5f)]
        public float randomPitch = 0.1f;

        public bool loop = false;

        private AudioSource source;

        public void SetSource(AudioSource _source)
        {
                source = _source;
                source.clip = clip;
                source.loop = loop;
        }

        public void Play()
        {
                source.volume = volume * (1 + Random.Range(-randomVolume / 2f, ran-
domVolume / 2f));
                source.pitch = pitch * (1 + Random.Range(-randomPitch / 2f, randomPitch
/ 2f));
                source.Play();
        }

        public void Stop()
        {
                source.Stop();
        }

}

public class AudioManager : MonoBehaviour
{

        public static AudioManager instance;

        [SerializeField]
        Sound[] sounds;

        void Awake()
        {
                if (instance != null)
                {
                        if (instance != this)
                        {
                                Destroy(this.gameObject);
```

```
                }
            }
            else
            {
                instance = this;
                DontDestroyOnLoad(this);
            }
        }

        void Start()
        {
            for (int i = 0; i < sounds.Length; i++)
            {
                GameObject _go = new GameObject("Sound_" + i + "_" +
sounds[i].name);
                _go.transform.SetParent(this.transform);
                sounds[i].SetSource(_go.AddComponent<AudioSource>());
            }

            PlaySound("Music");
        }

        public void PlaySound(string _name)
        {
            for (int i = 0; i < sounds.Length; i++)
            {
                if (sounds[i].name == _name)
                {
                    sounds[i].Play();
                    return;
                }
            }

            // no sound with _name
            Debug.LogWarning("AudioManager: Sound not found in list, " + _name);
        }

        public void StopSound(string _name)
        {
            for (int i = 0; i < sounds.Length; i++)
            {
                if (sounds[i].name == _name)
                {
                    sounds[i].Stop();
                    return;
                }
            }

            // no sound with _name
            Debug.LogWarning("AudioManager: Sound not found in list, " + _name);
        }
```

```
}
```

Файл MenuManager.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MenuManager : MonoBehaviour {

	[SerializeField]
	string hoverOverSound = "ButtonHover";

	[SerializeField]
	string pressButtonSound = "ButtonPress";

	AudioManager audioManager;

	void Start ()
	{
		audioManager = AudioManager.instance;
		if (audioManager == null)
		{
			Debug.LogError("No audiomanager found!");
		}
	}

	public void StartGame ()
	{
		audioManager.PlaySound(pressButtonSound);

		SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex +
1);
	}

	public void QuitGame()
	{
		audioManager.PlaySound(pressButtonSound);

		Debug.Log("WE QUIT THE GAME!");
		Application.Quit();
	}

	public void OnMouseOver ()
	{
		audioManager.PlaySound(hoverOverSound);
	}

}
```

Файл Weapon.cs

```
using UnityEngine;
using System.Collections;

public class Weapon : MonoBehaviour {
```

```
	public float fireRate = 0;
	public int Damage = 10;
	public LayerMask whatToHit;

	public Transform BulletTrailPrefab;
	public Transform HitPrefab;
	public Transform MuzzleFlashPrefab;
	float timeToSpawnEffect = 0;
	public float effectSpawnRate = 10;

	// Handle camera shaking
	public float camShakeAmt = 0.05f;
	public float camShakeLength = 0.1f;
	CameraShake camShake;

	public string weaponShootSound = "DefaultShot";

	float timeToFire = 0;
	Transform firePoint;

	// Caching
	AudioManager audioManager;

	// Use this for initialization
	void Awake () {
		firePoint = transform.FindChild ("FirePoint");
		if (firePoint == null) {
			Debug.LogError ("No firePoint? WHAT?!");
		}
	}

	void Start()
	{
		camShake = GameMaster.gm.GetComponent<CameraShake>();
		if (camShake == null)
			Debug.LogError("No CameraShake script found on GM object.");

		audioManager = AudioManager.instance;
		if (audioManager == null)
		{
			Debug.LogError("FREAK OUT! No audiomanager found in
scene.");
		}
	}

	// Update is called once per frame
	void Update () {
		if (fireRate == 0) {
			if (Input.GetButtonDown ("Fire1")) {
				Shoot();
			}
		}
```

```
		else {
			if (Input.GetButton ("Fire1") && Time.time > timeToFire) {
				timeToFire = Time.time + 1/fireRate;
				Shoot();
			}
		}
	}

	void Shoot () {
		Vector2 mousePosition = new Vector2 (Camera.main.Screen-
ToWorldPoint (Input.mousePosition).x, Camera.main.ScreenToWorldPoint(In-
put.mousePosition).y);
		Vector2 firePointPosition = new Vector2 (firePoint.position.x, fire-
Point.position.y);
		RaycastHit2D hit = Physics2D.Raycast (firePointPosition, mousePosition-
firePointPosition, 100, whatToHit);

		Debug.DrawLine (firePointPosition, (mousePosition-firePointPosi-
tion)*100, Color.cyan);
		if (hit.collider != null) {
			Debug.DrawLine (firePointPosition, hit.point, Color.red);
			Enemy enemy = hit.collider.GetComponent<Enemy>();
			if (enemy != null) {
				enemy.DamageEnemy (Damage);
				//Debug.Log ("We hit " + hit.collider.name + " and did " +
Damage + " damage.");
			}
		}

		if (Time.time >= timeToSpawnEffect)
		{
			Vector3 hitPos;
			Vector3 hitNormal;

			if (hit.collider == null) {
				hitPos = (mousePosition - firePointPosition) * 30;
				hitNormal = new Vector3(9999, 9999, 9999);
			}
			else
			{
				hitPos = hit.point;
				hitNormal = hit.normal;
			}

			Effect(hitPos, hitNormal);
			timeToSpawnEffect = Time.time + 1 / effectSpawnRate;
		}
	}

  void Effect(Vector3 hitPos, Vector3 hitNormal)
  {
```

```
            Transform trail = Instantiate (BulletTrailPrefab, firePoint.position, firePoint.rotation) as Transform;
                LineRenderer lr = trail.GetComponent<LineRenderer>();

                if (lr != null)
                {
                    lr.SetPosition(0, firePoint.position);
                    lr.SetPosition(1, hitPos);
                }

                Destroy(trail.gameObject, 0.04f);

                if (hitNormal != new Vector3(9999, 9999, 9999))
                {
                    Transform hitParticle = Instantiate(HitPrefab, hitPos, Quaternion.FromToRotation (Vector3.right, hitNormal)) as Transform;
                    Destroy(hitParticle.gameObject, 1f);
                }

        Transform clone = Instantiate (MuzzleFlashPrefab, firePoint.position, firePoint.rotation) as Transform;
                clone.parent = firePoint;
                float size = Random.Range (0.6f, 0.9f);
                clone.localScale = new Vector3 (size, size, size);
                Destroy (clone.gameObject, 0.02f);

                //Shake the camera
                camShake.Shake(camShakeAmt, camShakeLength);

                //Play shoot sound
                audioManager.PlaySound(weaponShootSound);
            }
}
```

Файл GameoverUI.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameOverUI : MonoBehaviour {

    [SerializeField]
    string mouseHoverSound = "ButtonHover";

    [SerializeField]
    string buttonPressSound = "ButtonPress";

    AudioManager audioManager;

    void Start ()
    {
        audioManager = AudioManager.instance;
        if (audioManager == null)
        {
```

```
                Debug.LogError("FREAK OUT! No AudioManager found in the
        scene.");
            }
        }

        public void Quit ()
        {
            audioManager.PlaySound(buttonPressSound);

            Debug.Log("APPLICATION QUIT!");
            Application.Quit();
        }

        public void Retry ()
        {
            audioManager.PlaySound(buttonPressSound);

            SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
        }

        public void OnMouseOver ()
        {
            audioManager.PlaySound(mouseHoverSound);
        }
```

Файл LiveCounterUI.cs

```
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(Text))]
public class LivesCounterUI : MonoBehaviour {

    private Text livesText;

    void Awake () {
        livesText = GetComponent<Text>();
    }

    // Update is called once per frame
    void Update () {
        livesText.text = "LIVES: " + GameMaster.RemainingLives.ToString();
    }
}
```

Файл WaveUI.cs

```
using UnityEngine;
using UnityEngine.UI;

public class WaveUI : MonoBehaviour {

    [SerializeField]
    WaveSpawner spawner;
```

```
    [SerializeField]
    Animator waveAnimator;

    [SerializeField]
    Text waveCountdownText;

    [SerializeField]
    Text waveCountText;

    private WaveSpawner.SpawnState previousState;

    // Use this for initialization
    void Start () {
        if (spawner == null)
        {
            Debug.LogError("No spawner referenced!");
            this.enabled = false;
        }
        if (waveAnimator == null)
        {
            Debug.LogError("No waveAnimator referenced!");
            this.enabled = false;
        }
        if (waveCountdownText == null)
        {
            Debug.LogError("No waveCountdownText referenced!");
            this.enabled = false;
        }
        if (waveCountText == null)
        {
            Debug.LogError("No waveCountText referenced!");
            this.enabled = false;
        }
    }

    // Update is called once per frame
    void Update () {
        switch (spawner.State)
        {
            case WaveSpawner.SpawnState.COUNTING:
                UpdateCountingUI();
                break;
            case WaveSpawner.SpawnState.SPAWNING:
                UpdateSpawningUI();
                break;
}

        previousState = spawner.State;
    }

    void UpdateCountingUI ()
```

```
        {
            if (previousState != WaveSpawner.SpawnState.COUNTING)
            {
                waveAnimator.SetBool("WaveIncoming", false);
                waveAnimator.SetBool("WaveCountdown", true);
                //Debug.Log("COUNTING");
            }
            waveCountdownText.text = ((int)spawner.WaveCountdown).ToString();
        }

        void UpdateSpawningUI()
        {
            if (previousState != WaveSpawner.SpawnState.SPAWNING)
            {
                waveAnimator.SetBool("WaveCountdown", false);
                waveAnimator.SetBool("WaveIncoming", true);

                waveCountText.text = spawner.NextWave.ToString();

                //Debug.Log("SPAWNING");
            }
        }
}
```

Файл StatusIndicator.cs

```
using UnityEngine;
using UnityEngine.UI;

public class StatusIndicator : MonoBehaviour {

    [SerializeField]
    private RectTransform healthBarRect;
    [SerializeField]
    private Text healthText;

    void Start()
    {
        if (healthBarRect == null)
        {
            Debug.LogError("STATUS INDICATOR: No health bar object
referenced!");
        }
        if (healthText == null)
        {
            Debug.LogError("STATUS INDICATOR: No health text object
referenced!");
        }
    }

    public void SetHealth(int _cur, int _max)
    {
        float _value = (float)_cur / _max;
```

```
                healthBarRect.localScale = new Vector3(_value, healthBarRect.lo-
        calScale.y, healthBarRect.localScale.z);
                healthText.text = _cur + "/" + _max + " HP";
        }

}
```

Файл EnemyAI.cs

```
using UnityEngine;
using System.Collections;
using Pathfinding;

[RequireComponent (typeof (Rigidbody2D))]
[RequireComponent (typeof (Seeker))]
public class EnemyAI : MonoBehaviour {

        // What to chase?
        public Transform target;

        // How many times each second we will update our path
        public float updateRate = 2f;

        // Caching
        private Seeker seeker;
        private Rigidbody2D rb;

        //The calculated path
        public Path path;

        //The AI's speed per second
        public float speed = 300f;
        public ForceMode2D fMode;

        [HideInInspector]
        public bool pathIsEnded = false;

        // The max distance from the AI to a waypoint for it to continue to the next way-
        point
        public float nextWaypointDistance = 3;

        // The waypoint we are currently moving towards
        private int currentWaypoint = 0;

        void Start () {
                seeker = GetComponent<Seeker>();
                rb = GetComponent<Rigidbody2D>();

                if (target == null) {
                        Debug.LogError ("No Player found? PANIC!");
                        return;
                }
```

```
		// Start a new path to the target position, return the result to the OnPath-
Complete method
		seeker.StartPath (transform.position, target.position, OnPathComplete);

		StartCoroutine (UpdatePath ());
	}

	IEnumerator UpdatePath () {
		if (target == null) {
			//TODO: Insert a player search here.
			return false;
		}

		// Start a new path to the target position, return the result to the OnPath-
Complete method
		seeker.StartPath (transform.position, target.position, OnPathComplete);

		yield return new WaitForSeconds ( 1f/updateRate );
		StartCoroutine (UpdatePath());
	}

	public void OnPathComplete (Path p) {
		Debug.Log ("We got a path. Did it have an error? " + p.error);
		if (!p.error) {
			path = p;
			currentWaypoint = 0;
		}
	}

	void FixedUpdate () {
		if (target == null) {
			//TODO: Insert a player search here.
			return;
		}

		//TODO: Always look at player?

		if (path == null)
			return;

		if (currentWaypoint >= path.vectorPath.Count) {
			if (pathIsEnded)
				return;

			Debug.Log ("End of path reached.");
			pathIsEnded = true;
			return;
		}
		pathIsEnded = false;

		//Direction to the next waypoint
```

```
                Vector3 dir = ( path.vectorPath[currentWaypoint] - transform.position
        ).normalized;
                dir *= speed * Time.fixedDeltaTime;

                //Move the AI
                rb.AddForce (dir, fMode);

                float dist = Vector3.Distance (transform.position, path.vectorPath[current-
        Waypoint]);
                if (dist < nextWaypointDistance) {
                        currentWaypoint++;
                        return;
                }
        }

}
```

Файл Fading.cs

```
        using UnityEngine;
        using System.Collections;

        public class Fading : MonoBehaviour {

                public Texture2D fadeOutTexture;   // the texture that will overlay the screen.
        This can be a black image or a loading graphic
                public float fadeSpeed = 0.8f;                  // the fading speed

                private int drawDepth = -1000;                  // the texture's order in the draw hier-
        archy: a low number means it renders on top
                private float alpha = 1.0f;                     // the texture's alpha value between 0
        and 1
                private int fadeDir = -1;                       // the direction to fade: in = -1 or out
        = 1

                void OnGUI()
                {
                        // fade out/in the alpha value using a direction, a speed and Time.del-
        taTime to convert the operation to seconds
                        alpha += fadeDir * fadeSpeed * Time.deltaTime;
                        // force (clamp) the number to be between 0 and 1 because GUI.color uses
        Alpha values between 0 and 1
                        alpha = Mathf.Clamp01(alpha);

                        // set color of our GUI (in this case our texture). All color values remain
        the same & the Alpha is set to the alpha variable
                        GUI.color = new Color (GUI.color.r, GUI.color.g, GUI.color.b, alpha);
                        GUI.depth = drawDepth;
                                                                // make the black tex-
        ture render on top (drawn last)
                        GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height),
        fadeOutTexture);                // draw the texture to fit the entire screen area
                }
```

```
        // sets fadeDir to the direction parameter making the scene fade in if -1 and out if
    1
        public float BeginFade (int direction)
        {
            fadeDir = direction;
            return (fadeSpeed);
        }

        // OnLevelWasLoaded is called when a level is loaded. It takes loaded level index
    (int) as a parameter so you can limit the fade in to certain scenes.
        void OnLevelWasLoaded()
        {
            // alpha = 1;          // use this if the alpha is not set to 1 by default
            BeginFade(-1);              // call the fade in function
        }
}
```

Файл Wavespawner.cs

```
using UnityEngine;
using System.Collections;

public class WaveSpawner : MonoBehaviour {

    public enum SpawnState { SPAWNING, WAITING, COUNTING };

    [System.Serializable]
    public class Wave
    {
        public string name;
        public Transform enemy;
        public int count;
        public float rate;
    }

    public Wave[] waves;
    private int nextWave = 0;
    public int NextWave
    {
        get { return nextWave + 1; }
    }

    public Transform[] spawnPoints;

    public float timeBetweenWaves = 5f;
    private float waveCountdown;
    public float WaveCountdown
    {
        get { return waveCountdown; }
    }

    private float searchCountdown = 1f;

    private SpawnState state = SpawnState.COUNTING;
```

```
public SpawnState State
{
	get { return state; }
}

void Start()
{
	if (spawnPoints.Length == 0)
	{
		Debug.LogError("No spawn points referenced.");
	}

	waveCountdown = timeBetweenWaves;
}

void Update()
{
	if (state == SpawnState.WAITING)
	{
		if (!EnemyIsAlive())
		{
			WaveCompleted();
		}
		else
		{
			return;
		}
	}

	if (waveCountdown <= 0)
	{
		if (state != SpawnState.SPAWNING)
		{
			StartCoroutine( SpawnWave ( waves[nextWave] ) );
		}
	}
	else
	{
		waveCountdown -= Time.deltaTime;
	}
}

void WaveCompleted()
{
	Debug.Log("Wave Completed!");

	state = SpawnState.COUNTING;
	waveCountdown = timeBetweenWaves;

	if (nextWave + 1 > waves.Length - 1)
	{
		nextWave = 0;
```

```
                Debug.Log("ALL WAVES COMPLETE! Looping...");
            }
            else
            {
                nextWave++;
            }
        }

        bool EnemyIsAlive()
        {
            searchCountdown -= Time.deltaTime;
            if (searchCountdown <= 0f)
            {
                searchCountdown = 1f;
                if (GameObject.FindGameObjectWithTag("Enemy") == null)
                {
                    return false;
                }
            }
            return true;
        }

        IEnumerator SpawnWave(Wave _wave)
        {
            Debug.Log("Spawning Wave: " + _wave.name);
            state = SpawnState.SPAWNING;

            for (int i = 0; i < _wave.count; i++)
            {
                SpawnEnemy(_wave.enemy);
                yield return new WaitForSeconds( 1f/_wave.rate );
            }

            state = SpawnState.WAITING;

            yield break;
        }

        void SpawnEnemy(Transform _enemy)
        {
            Debug.Log("Spawning Enemy: " + _enemy.name);

            Transform _sp = spawnPoints[ Random.Range (0, spawnPoints.Length) ];
            Instantiate(_enemy, _sp.position, _sp.rotation);
        }

}
```